AngularJS

Alejandro Such Berengier `<<alejandro.such@gmail.com>>`

Table of Contents

1. AngularJS

1.1. ¿Qué es AngularJS?

AngularJS es un framework de JavaScript de código abierto, mantenido por Google, que ayuda con la gestión de lo que se conoce como aplicaciones de una sola página (en inglés, single-page applications). Su objetivo es aumentar las aplicaciones basadas en navegador con (MVC) Capacidad de Modelo Vista Controlador, en un esfuerzo para hacer que el desarrollo y las pruebas más fáciles.

AngularJS es un framework JavaScript, mantenido por Google, que se ejecuta en el lado del cliente, y se utiliza para crear *single-page web applications*.

¿Single-page web applications?

Una *single-page web application* (en adelante SPA), es una aplicación web que se ejecuta completamente en una única página web, con el objetivo de proporcionar una experiencia más fluida y similar a la que nos encontraríamos en una aplicación de escritorio

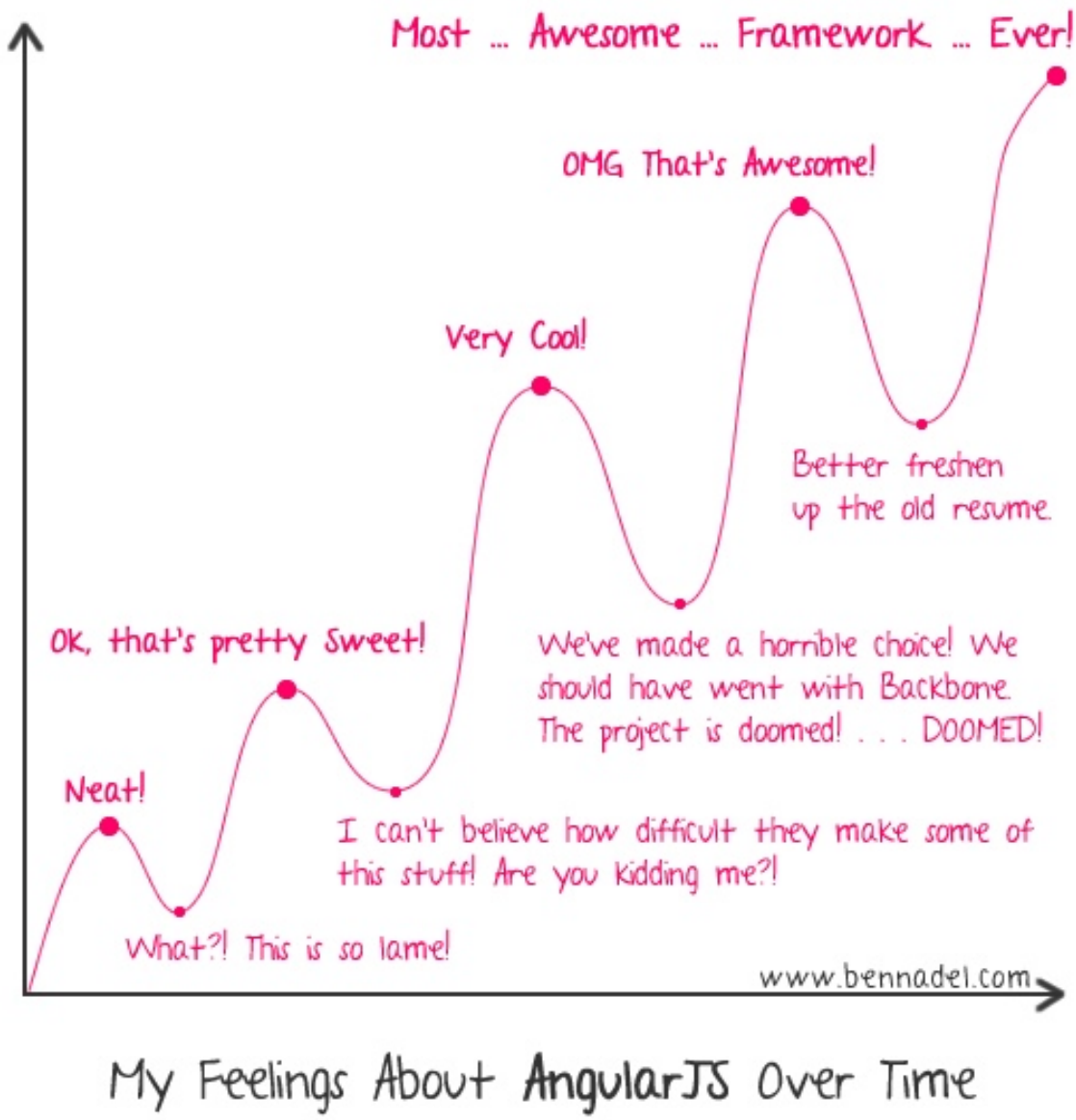
En una aplicación SPA, todos los datos necesarios, como el HTML, CSS o JavaScript, se cargan y añaden en la página cuando es necesario, normalmente respondiendo a acciones del usuario. En ningún momento del proceso veremos una recarga total de la página. Para esto, como os imaginaréis a lo largo del proceso de ejecución de una aplicación SPA existe una comunicación con el servidor en segundo plano.

Ejemplos de aplcaciones SPA

- GMail
- Google+
- Facebook
- Twitter
- App de YouTube de PS3, que además está realizada con AngularJS

1.2. Volviendo a AngularJS…

Volviendo a AngularJS y a sus características principales. AngularJS es un framework para escribir aplicaciones en JavaScript. Es prescriptivo, ya que existe una manera recomendada de hacer las cosas con él. Además, tiene su propia vuelta de tuerca del omnipresente patrón MVC, especialmente adaptado a JavaScript y al desarrollo de aplicaciones SPA.



1.3. Principales características de AngularJS

A continuación, vamos a ver alogos de los aspectos más importantes de AngularJS. Puede que ahora se mencionen otros conceptos, que en su momento quedarán aclarados.

Two-way data binding

Según Martin Fowler[1], este patrón consiste en: *Un mecanismo que garantiza que cualquier cambio realizado sobre los datos en un control de la interfaz, se traslada inmediatamente al estado de la sesión (y viceversa).*

En una aplicaciónAngularJS el *two-way data binding* consiste en la sincronización automática entre los componentes del modelo y de la vista. La vista es una proyección del modelo en todo momento. Cuando realizamos un cambio sobre el modelo éste se refleja inmediatamente sobre la vista. image::img/ses01/02.png[{large-size}]

[1] http://martinfowler.com/eaaDev/DataBinding.html

En AngularJS, este mecanismo funciona de la siguiente manera: una plantilla (que es código HTML) se compila en el navegador. Esta compilación hace que cualquier cambio en el modelo se refleje inmediatamente en la vista. También hará que todo cambio que realicemos en la vista se propage al modelo. El modelo es la *single-source-of-truth* del estado de la aplicación, lo que simplifica mucho las cosas, al no tener que gestionar más que el modelo, y pensar el la vista como una proyección de éste.

Dado que la vista es una proyección del modelo, el controlador queda totalmente separado de la vista y no es consciente de ella. De esta manera, realizar tests sobre un controlador es mucho más sencillo, ya que no depende de la vista ni de ningún elemento del DOM.

Imaginémonos que queremos implementar esta funcionalidad en JavaScript plano, o con jQuery. Tendríamos que hacer un número enorme de consultas al DOM para mantener esta funcionalidad, y hacer que se ejecute periódicamente para que los datos siempre se encuentren actualizados. Sin embargo, AngularJS ya realiza este trabajo por nosotros, y el *data binding* se realiza de manera transparente.

Así, para una pantalla que muestre el nombre de usuario, sería tan sencillo como declarar una variable con el nombre de usuario en nuestro código JavaScript:

```
var name = 'Alex';
```

En nuestra vista, utilizaremos la notación {{ }}, que nos pemite ligar expresiones a elementos:

```
<h1>Bienvenido, {{ name }}</h1>
```

MVW (Model-View-Whatever)

Esto es lo que piensa Igor Minar, *lead* de AngularJS, cuando se entra en cuestiones sobre qué patrón sigue el framework.

> MVC vs MVVM vs MVP. Un asunto controvertido en el que muchos desarrolladores pueden pasarse horas y horas discutiendo. Durante muchos años, AngularJS estaba más cerca del MVC que de cualquier otra cosa. Sin embargo, a medida que pasó el tiempo y debifo a una serie de refactorings y mejoras en la API, ahora está más cerca de MVVM, donde el objeto $scope puede considerarse como un ViewModel que podemos decorar con una función que llamamos Controlador. Ser capaces de categorizar un framework como un MV* tiene sus ventajas. Ayuda a los desarrolladores a estar más cómodos con sus APIS, haciendo más fácil crear un modelo mental que representa la aplicación que están construyendo con el framework. También, ayuda a establecer cierta terminología a emplear por los desarrolladores. Una vez dicho esto, veo muchos desarrolladores que construyen aplicaciones alucinantes, muy bien diseñadas, que siguen los principios de *separation of concerns*; y muy pocos que pierden el tiempo discutiendo chorradas dobre MV*. Y por esta razón, por la presente declaro que AngularJS es un framework MVW (Model-View-Whatever). Donde *whatever* es *whatever works for you*. Angular proporciona mucha flexibilidad para separar de manera sencilla la lógica de presentación de la de diseño y el estado de presentación. Por favor, úsalo para incrementar tu productividad y mantenibilidad de tu aplicación, y no para discutir sobre cosas que, en el fondo, no importan tanto.

— Igor Minar *https://plus.google.com/+AngularJS/posts/aZNVhj355G2*

A efectos prácticos, y como se dice en la cita, el patrón *Model View ViewModel* (MVVM) es una aproximación bastante cercana para describir de manera general el comportamiento de AngularJS.

El patrón MVVM funciona muy bien en aplicaciones con interfaces de usuario ricas, ya que la Vista se vincula al ViewModel y, cuando el estado del ViewModel cambia, la Vista se actualiza automáticamente gracias, precisamente, al *two-way-databinding*. En AngularJS, una Vista es simplemente código HTML compilado con elementos propios del framework. Una vez finaliza el ciclo de compilación, la Vista se vincula al objeto $scope, que es el ViewModel. Ya veremos en profundidad el objeto $scope, pero adelantamos que es un objeto JavaScript que captura ciertos eventos para permitir el *data binding*. También, podemos exponer funciones al ViewModel para poder ejecutar funciones.

image::img/ses01/003_mvvm.png

Ya veremos esto más adelante, y veremos lo sencillo que es trabajar con este patrón, que crea una separación muy clara entre la Vista y la lógica que la conduce. Uno de los "efectos secundarios" del patrón ViewModel, es que permite que el código sea muy testable.

Plantillas HTML

Otra de las grandes características de AngularJS es el uso de HTML para la creación de plantillas. Éstas pueden ser útiles cuando queremos predefinir un layout con secciones dinámicas está conectado con una estructura de datos. Por ejemplo, para repetir un mismo elemento DOM en una página, como una lista o una tabla. Podríamos definir cómo queremos que se vea una fila, y después asociarle una estructura de datos, como un array de JavaScript. Esta plantilla se repetiría tantas veces como ítems encontremos en el array, asociando el ítem al contenido.

Hay muchas librerías de *templating*, y muy buenas. Pero para la mayoría de ellas se requiere aprender una nueva sintaxis. Esta complejidad adicional puede ralentizar al desarrollador. Además, muchas suelen pasar por un preprocesador.

El navegador evalúa las plantillas de AngularJS como el resto de HTML de la página. Hay cierta funcionalidad de AngularJS que gestiona cómo se representan los datos, que veremos cómo funciona en las próximas sesiones.

Deep linking

Como hemos comoentado, AngularJS es un framework para construir aplicaciones SPA. Sin embargo, es posible que nuestros usuarios no se den cuenta de este detalle. En muchas aplicaciones SPA modernas, está prohibido usar el *back button* ya que no se tiene en cuenta a la hora de programar. Sin embargo, AngularJS hace uso de la API de history de HTML5. ¿Que tu navegador no implementa esa API? Es igual, AngularJS seguirá gestionando bien el histórico gracias al control de cambios en el hashbang.

Para un usuario, esto significa que se pueden guardar y compartir cualquier estado de la apliación, cosa muy importante hoy en día debido al social media. También nos permite a los desarrolladores cambiar el estado de la aplicación de la manera más sencilla posible: mediante el uso de hipervínculos.

Inyección de dependencias

La inyección de dependencias (DI por sus siglas en inglés) describe una técnica que hemos estado usando toda la vida. Si alguna vez has usado una función que acepta un parámetro, ya has hecho uso de la inyección de dependencias. Inyectas algo de lo que depende tu función para realizar su trabajo. Nada más y nada menos.

image::img/ses01/004_di.png

Al modularizar tu código con elementos inyectables, éste es más fácil de testear, ya que en cualquier momento puedes reemplazar uno de los elementos por otro, siempre y cuando implemente la misma interfaz.

Veamos un sencillo caso de inyección de dependencias. El siguiente código no hace uso de ellas:

```
function a () {
 return 5;
}
function b () {
 return a() + 1;
}
console.log(b());
```

Con un código tan pequeño y tan simple, ya se plantean dos problemas:

- Orden: necesariamente, la función *a* debe cargarse antes que *b*. Si están en ficheros separados, hay que tener esto en cuenta. ¿Qué pasa cuando nuestra aplicación crece más y más? Nuestra aplicación se hace dependiente del orden.
- Testabilidad: Se hace imposible sustituir *a* por un objeto mock para poder testarlo.

Si usáramos DI, nuestro código tendría un aspecto similar al siguiente:

```
service('a', function () {
 return 5;
});
service('b', function (a) {
 return a() + 5;
});
service('main', function (b) {
 console.log(b());
});
```

Este cambio tiene varias ventajas: por una parte, ya no dependemos del orden, y nuestro código no tiene que seguir una secuencia. estp implica que, podemos extraer cualquiera de los bloques a otro fichero, lo que en aplicaciones grandes será más que conveniente. Además, en cualquier momento podemos sobreescribir cualquiera de las funciones, cosa muy importante para realizar tests.

Directivas

Las directivas son la parte más interesante de AngularJS, ya que nos permiten extender HTML para que realice todo lo que nosotros queramos.

Podemos crear elementos del DOM personalizados, atributos o clases que incorporan cierta funcionalidad definida en JavaScript. Aunque HTML es excelente para definir un layout, para el resto se queda corto. Las directivas nos proporcionan un mecanismo para unir la naturaleza declarativa de HTML con la naturaleza funcional de JavaScript en un mismo elemento. Así, cuando aprendamos a utilizar directivas, en lugar de pintar un modal de bootstrap de esta manera:

```
<div class="modal fade">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal" aria-
hidden="true">&times;</button>
        <h4 class="modal-title">Esto sería un modal con bootstrap y
 jQuery</h4>
      </div>
      <div class="modal-body">He aquí el contenido del modal</div>
    </div>
  </div>
</div>
```

Lo haremos de esta otra:

```
<modal title="Esto sería un modal con AngularJS">He aquí el contenido del
 modal. Fácil, ¿no?</modal>
```

Para los curiosos, podéis ir jugando con el código de este modal en http://jsfiddle.net/alexsuch/RLQhh/.

1.4. Ventajas e inconvenientes de AngularJS

Se ha escrito mucho acerca de las ventajas e inconvenientes de este framework. En este apartado hacemos una recopilación de los elementos más destacados y odiados por la comunidad.

Ventajas

- Es comprensivo. Se trata de un framework con una curva de aprendizaje poco elevada, lo que nos permite estar desarrollando aplicaciones sencillas en poco tiempo. No necesitamos conocer a fondo los conceptos de módule, service, factory, scope, inyección de dependencias o directive. ¡Con saber hacer un controlador y las directivas que proporciona el sistema ya podemos echar a correr!.
- Trabajamos con POJSOS (Plain Old Javascript Objects). Se puede, y se recomienda, el uso de primitivas y objetos Javascript como Arrays, Dates, y objetos todo lo complejos que queramos para trabajar con AngularJS.
- Inyección de depenedencias. Como hemos visto, la inyección de dependencias nos permite desacoplar nuestro código, y hacerlo más testable.
- Las plantillas pueden escribirse en HTML. AngularJS es muy bueno para la construcción de single page applications, y eso se nota en la facilidad para la creación de plantillas.

A diferencia de otros frameworks, donde las plantillas se definen en variables o con una sintaxis específica, en AngularJS podemos definir plantillas utilizando exclusivamente HTML y documentos HTML, que podemos extender o no mediante directivas.

- Muy buena integración del framework con REST y AJAX. De manera que con muy pocas líneas de código podemos realizar una petición al servidor, obtener y procesar los datos, y mostrar la información que necesitamos en nuestra vista.
- Integración con jQuery. AngularJS trabaja de base con un subset de jQuery, llamado jqLite, que permite la manipulación del DOM de manera compatible en todos los navegadores. Todas las referencias a elementos de angular están envueltos con jqLite. Sin embargo, si añadimos jQuery a nuestro proyecto, el framework lo detecta y hace uso de éste en lugar de jQuery.
- Modularidad. Se pueden elaborar de manera sencilla componentes que podremos reutilizar en posteriores desarrollos.
- Minificación. La Minificación tiene un doble objetivo: por un lado, conseguir que nuestra aplicación “pese” menos; por otro, se pretende ofuscar de alguna manera el código para hacer un poco más segura nuestra aplicación. AngularJS tiene en cuenta esto, y nos permite minificar nuestra aplicación de manera sencilla y con pocos cambios, en caso de no haberlo previsto inicialmente.

Inconvenientes

- Al igual que hemos dicho que la curva de aprendizaje inicial es muy baja, cuando queremos hacer algo avanzado en aplicaciones más serias, la cosa puede resultar un poco difícil debido a la falta de documentación, o documentación errónea en algunos casos. Afortunadamente, el equipo de AngularJS está trabajando a diario en esto, y entre esto y la cada vez más creciente comunidad de usuarios, este problema se va minificando a la carrera.
- El framework de validación de formularios no es del todo perfecto, y de vez en cuando hay que hacer algunos trucos para que haga lo que nosotros queremos.
- Como pasa con todos los frameworks, ninguno es la panacea y encontraremos escenarios donde AngularJS no encaje.

1.5. Recursos online

AngularJS tiene una comunidad de desarrolladores cada vez más extensa. Algu nos de los canales que podemos utilizar para discutir sobre cualquier elemento del framework o pedir ayuda son:

- Lista de correo en angular@googlegroups.com[2]
- Comunidad de Google+ https://plus.google.com/u/0/communities/115368820700870330756
- Canal de IRC #angularjs
- Tag *angularjs* en http://stackoverflow.com
- Twitter @angularjs

[2] mailto:angular@googlegroups.com

- http://builtwith.angularjs.org/ nos proporciona un listado de grandes aplicaciones realizadas con AngularJS

Además, tenemos diversos sitios donde poder seguir con el aprendizaje de AngularJS, como pueden ser:

- Página oficial de AngularJS (http://www.angularjs.org). Donde tenemos un poco de todo: videotutoriales, referencia de la API, posibilidad de descargarnos el código fuente, etc. Un must have en nuestros marcadores del navegador.
- Blog de AngularJS: http://blog.angularjs.org
- Canal de youtube de AngularJS (http://www.youtube.com/user/angularjs) donde hay charlas, tutoriales y una variedad muy extensa de vídeos.
- http://egghead.io/. Otro *must have*, y probablemente uno de los mejores sitios de videotutoriales de AngularJS
- Blog de Alejandro Such (http://veamospues.wordpress.com). Escribe poco, pero cuando lo hace es muy interesante ;)

También, hacer especial mención a la web http://ngmodules.org/. En ella encontramos una base de datos con un extenso repertorio de módulos hechos por la comunidad y que pueden servirnos en muchos de nuestros proyectos. Encontramos módulos tan usados como angular-ui o angulartics. Cómo no, siempre es recomendable pasarse de vez en cuando por github a ver si hay proyectos similares a módulos que podamos necesitar. <<<

2. Una tarde con AngularJS

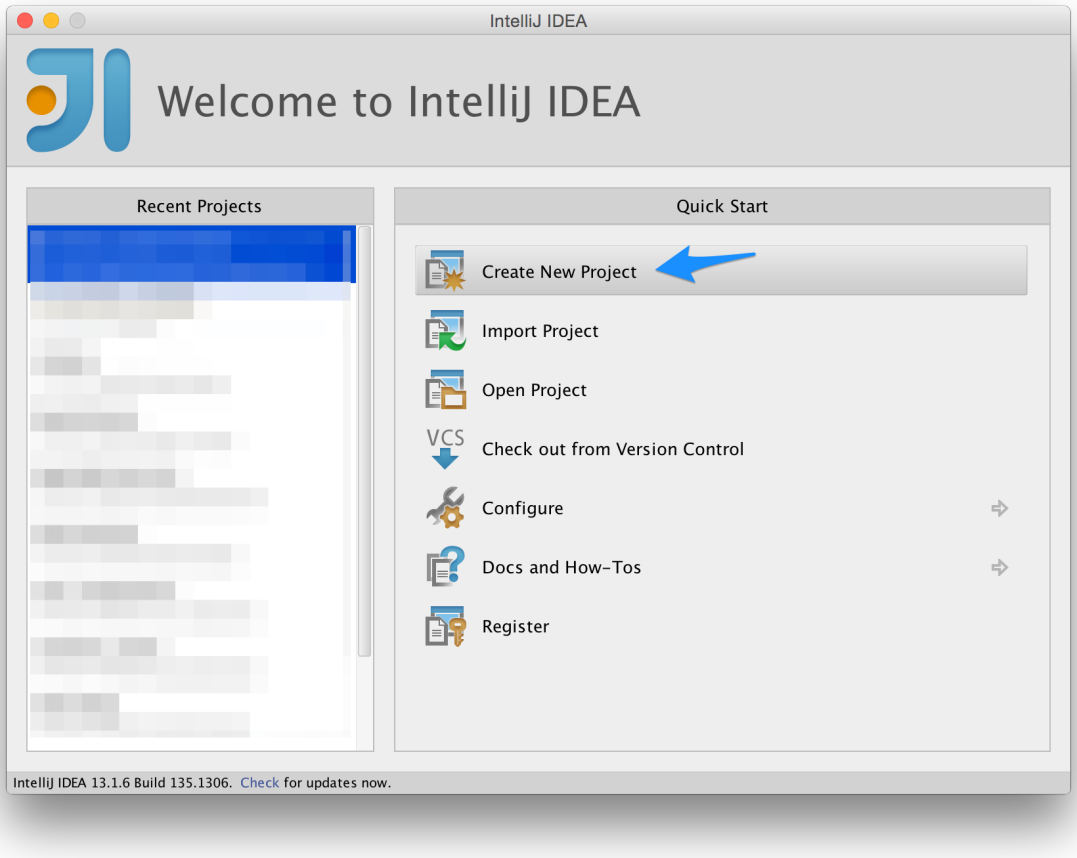
En este capítulo vamos a ver todos los fundamentos de AngularJS para ponernos a trabajar rápidamente con el Framework.

Una vez terminada la sesión, seremos capaces de crear aplicaciones sencillas con AngularJS, y tendremos conocimiento de unos fundamentos que iremos desarrollando y ampliando en sesiones posteriores.

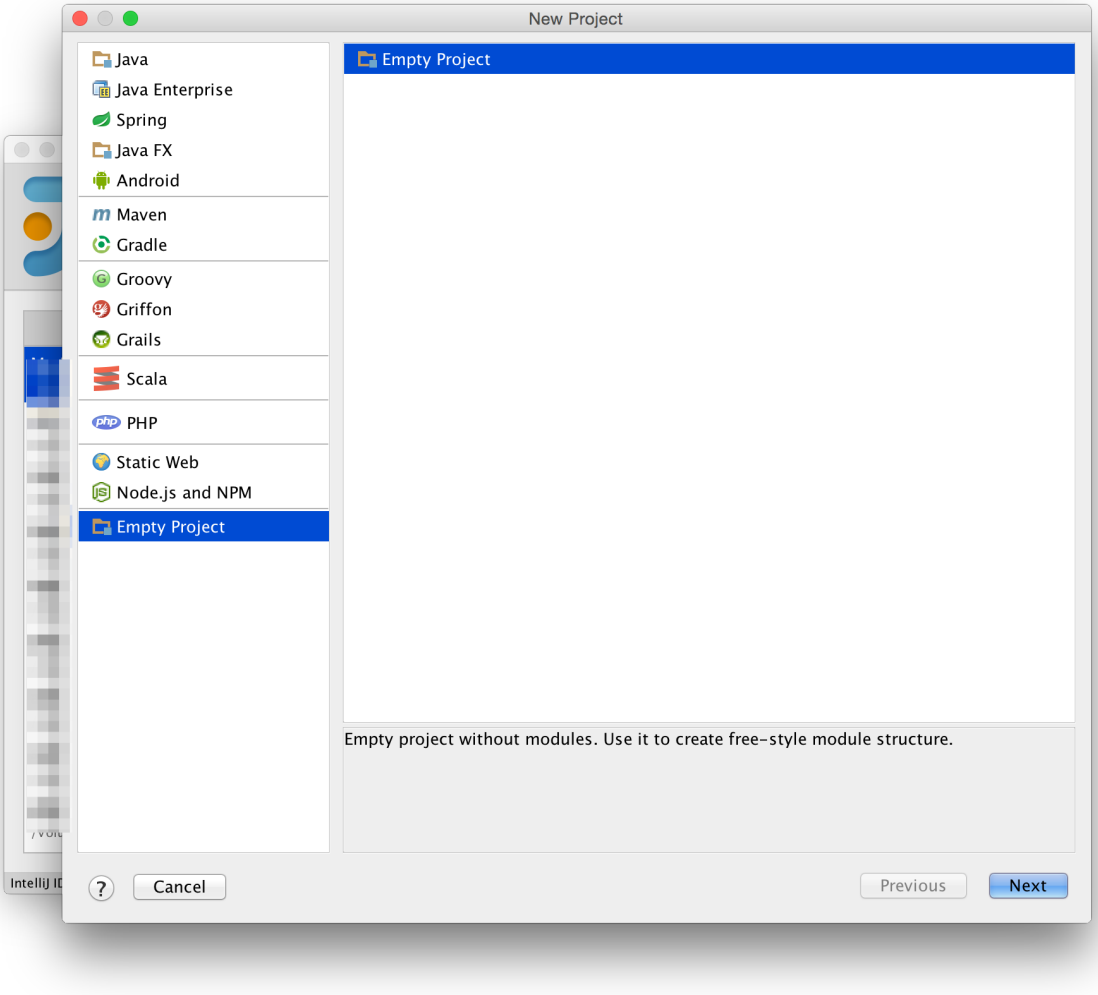
> Todo el código que hagamos de aquí en adelante lo pondremos en un *fork* del proyecto `java_ua/angular-expertojava`.
>
> Como hay una parte del código de los ejercicios que evolucionará a lo largo de las sesiones, identificaremos cada entrega mediante el uso de **tags**. El nombre de dichos tags se indicará en cada apartado de ejercicios.

2.1. Creando la aplicación

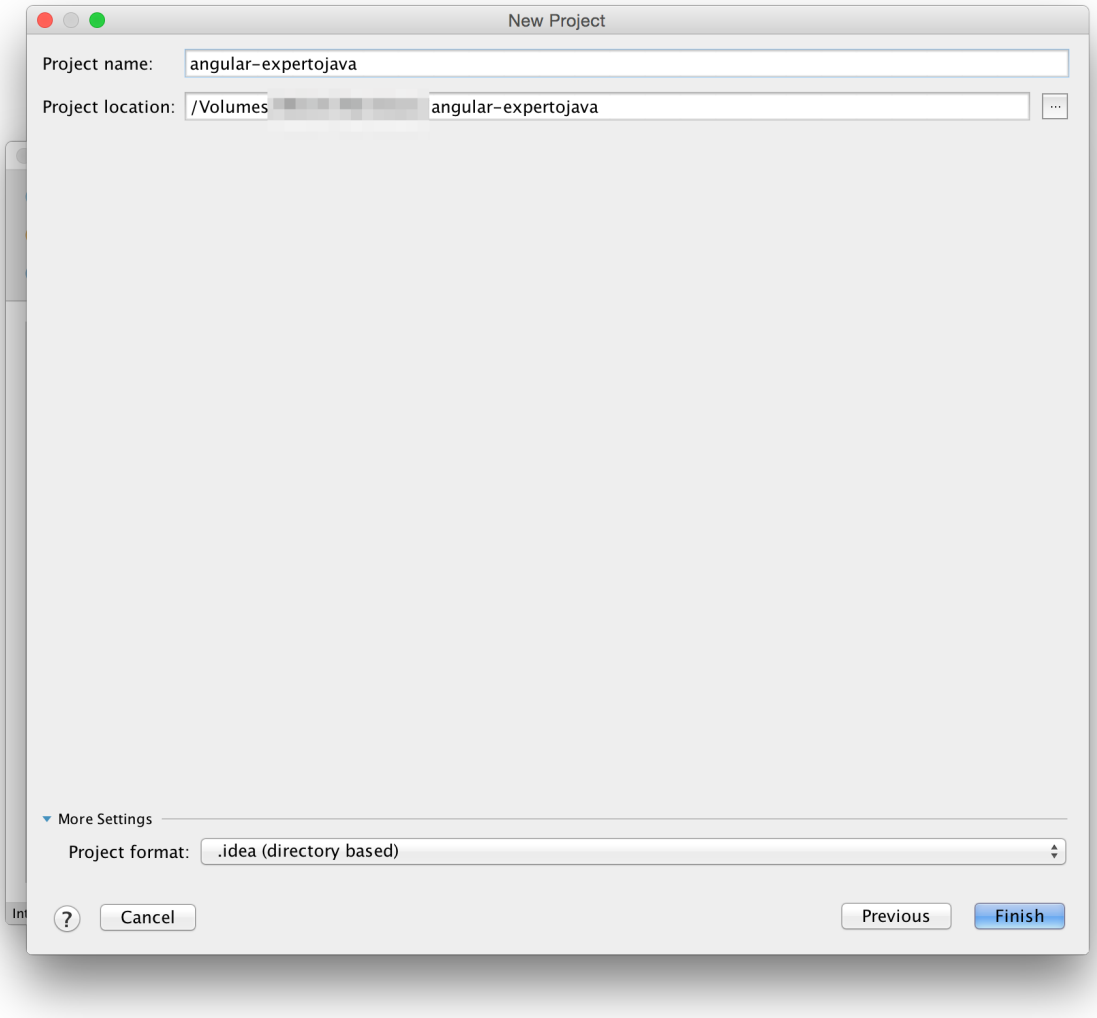
Para nuestra aplicación, crearemos un nuevo proyecto IntelliJ



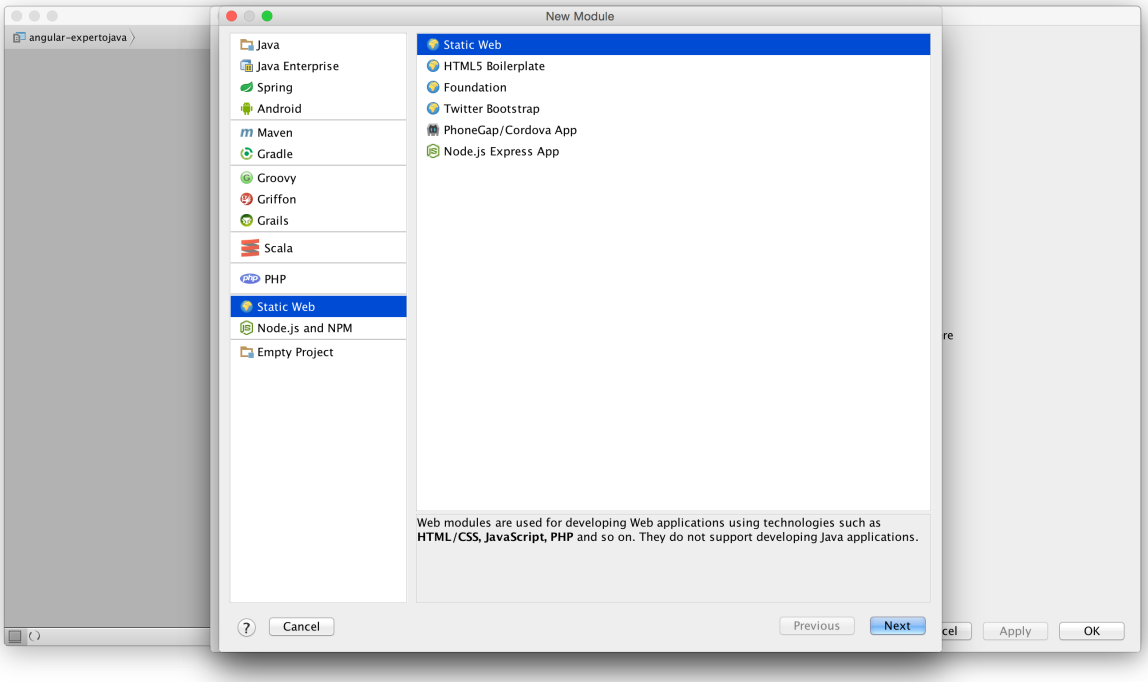
Crearemos un proyecto vacío, que será la base de todos nuestros módulos y será lo que subiremos a BitBucket

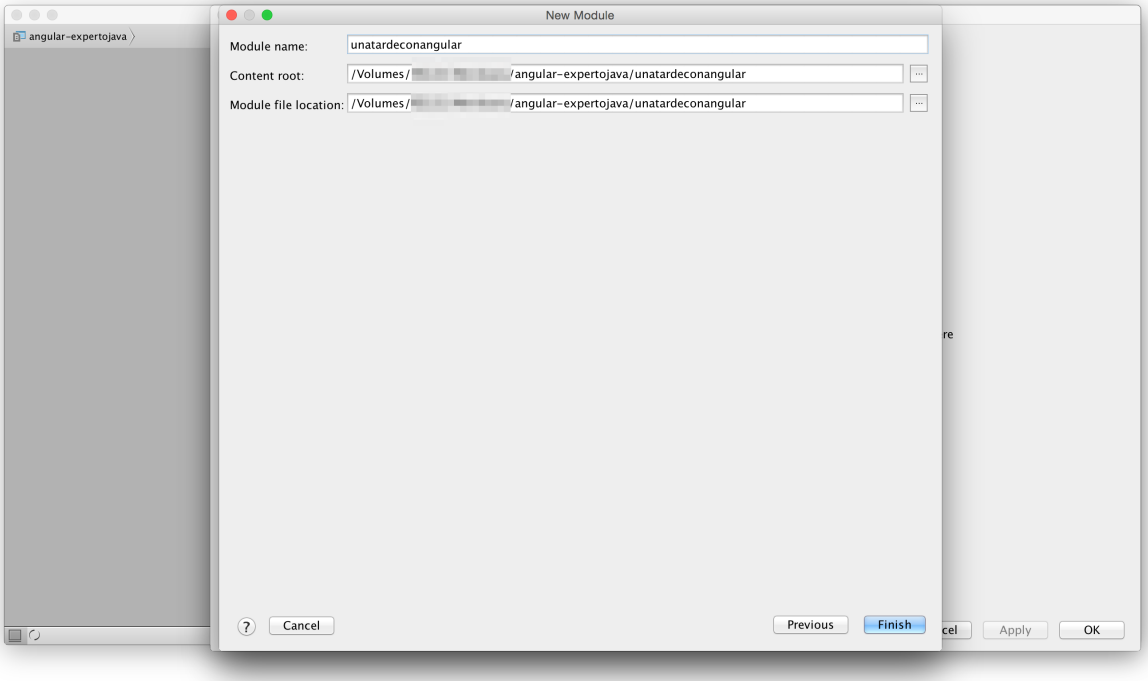


Llamaremos al proyecto `angular-expertojava`



A continuación, introduciremos un nuevo módulo de tipo *Static Web*. Lo llamaremos `unatardeconangular`





2.2. Directivas, data binding

Dado que AngularJS es un framework JavaScript se se va a ejecutar en una página web, vamos a necesitar un fichero html. En éste, tendremos que incluir el código fuente de AngularJS.

```
<!DOCTYPE html>
```

```
<html ng-app>
    <head>
        <title>Una tarde con AngularJS</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    </head>
    <body>
        <script src="https://code.angularjs.org/1.2.16/angular.min.js"></
script>
    </body>
</html>
```

En el momento de escribir estos apuntes, la última versión estable es la 1.2.16. Cualquier versión posterior, bien sea 1.2.x o la más reciente 1.3.x es totalmente compatible con lo que aquí veamos.

Una vez añadido el script de AngularJS en nuestra página, ya podemos empezar a usarlo. Para esto, nos vamos a topar con el primero de los elementos propios de AngularJS: las directivas. Una directiva, como se ha mencionado, nos permite extender HTML, creando componentes, clases o atributos con una funcionalidad dada.

Por ejemplo, en el código de arriba, ya vemos nuestra primera directiva, y una de las más importantes: `ng-app`. Por convenio, todo lo que tenga la forma `ng-*` va a ser una directiva *built in* de AngularJS. Dado que no son exclusivas del core del framework y nosotros podemos crearnos nuestras propias directivas, por convenio las *third parties* suelen utilizar un prefijo de 2-3 caracteres para cada directiva y así evitar conflictos de nombres.

Volviendo a la directiva `ng-app`. Ésta se encarga de inicializar nuestra aplicación. Lo más normal es ubicarla cerca del elemento raíz de la página, es decir, en las etiquetas `<body>` o `<html>`.

También podemos definir un módulo de AngularJS, que será nuestro módulo raíz para la aplicación. De momento, no le definiremos ningún nombre a la directiva ya que podemos hacer muchas cosas sin añadir ningún módulo. Por ejemplo, probemos el siguiente bloque de código:

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <title>Una tarde con AngularJS</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="stylesheet" href="components/lib/twitter-bootstrap/css/
bootstrap.css" />
    </head>
    <body>
        <div class="container">
            <label>Nombre:</label> <input type="text" ng-model="name" /
><br/>
            Has escrito {{ name }}.
        </div>
        <script src="https://code.angularjs.org/1.2.16/angular.min.js"></
script>
    </body>
```

```
</html>
```

Aquí vemos en acción otra directiva fundamental de AngularJS, llamada `ng-model`. Lo que hace esta directiva es crear una variable (en este caso llamada *name* porque estamos solicitando un nombre al usuario, pero podría ser *surname* o *phone*), en un elemento en memoria llamado `scope`. Este `scope` es un ViewModel que estaba vacío hasta que le hemos asignado esa propiedad. Para escribir ese valor del `scope` en cualquier sitio de la vista, lo único que tenemos que hacer es usar la expresión de *data binding* `{{ }}`.

Como algunos sabréis, la especificación de HTML5 dice que los *custom attributes* deben empezar con el prefijo `data-`. Si en algún proyecto os encontráis que hay que seguir esta especificación no os preocupéis: el core de AngularJS soporta este prefijo e identifica todas las directivas que lo lleven.

Si probamos el código en nuestro navegador, veremos cómo el nombre se va escribiendo a medida que introducimos caracteres en el input, gracias a la acción del *two way data binding*.

2.3. Declarando colecciones e iterando sobre ellas

Otra cosa que podemos hacer es recorrer una colección. Para ello, AngularJS nos proporciona una directiva, `ng-repeat`, que nos permite iterar sobre un array de elementos. Este array lo vamos a inicializar con otra directiva, llamada `ng-init`:

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <title>Una tarde con AngularJS</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="stylesheet" href="components/lib/twitter-bootstrap/css/
bootstrap.css" />
    </head>
    <body>
        <div class="container" ng-init="people = ['Domingo', 'Otto',
 'Aitor', 'Eli', 'Fran', 'José Luís', 'Alex']">
            <ul>
                <li ng-repeat="person in people">{{ person }}</li>
            </ul>
        </div>
        <script src="https://code.angularjs.org/1.2.16/angular.min.js"></
script>
    </body>
</html>
```

El uso de la directiva ng-init es muy similar a la declaración de variables dentro de código JavaScript. Por suerte, muy pronto veremos que no vamos a inicializar variables de esta manera, ya que como imaginaréis una vista no es el mejor lugar para inicializar una variable.

Por su parte, hemos visto que únicamente hemos declarado una plantilla para la colección, y ha sido la directiva `ng-repeat` quien se ha encargado de instanciarla para cada elemento de la colección. El formato de expresión más utilizado para esta directiva es `variable in expression`, donde `variable` es el nombre de una variable definida por el usuario,

y `expression` es el nombre de nuestra colección. Más adelante, veremos que hay más expresiones que podemos usar con esta directiva.

2.4. Filtros

Otra cosa que podemos usar en AngularJS son filtros. Un filtro se encarga de recibir entrada determinada, realizar una transformación sobre ella, y devolver el resultado de la misma.

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <title>Una tarde con AngularJS</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="stylesheet" href="components/lib/twitter-bootstrap/css/
bootstrap.css" />
    </head>
    <body>
        <div class="container" ng-init="people = [{name:'Domingo',
 subject:'JPA'}, {name:'Otto', subject:'Backbone'},
 {name:'Aitor',subject:'JavaScript'}, {name:'Miguel Ángel',subject:'JHD'},
 {name:'Eli', subject:'REST'}, {name:'Fran', subject:'Grails'},
 {name:'José Luís', subject:'PaaS'}, {name:'Alex', subject:'AngularJS'}]">
            <label>Filtro:</label> <input type="text" ng-
model="textFilter" /><br/>
            <ul>
                <li ng-repeat="person in people | filter:textFilter |
 orderBy:'name'">{{ person.name }} - {{ person.subject | uppercase }}</li>
            </ul>
        </div>
        <script src="https://code.angularjs.org/1.2.16/angular.min.js"></
script>
    </body>
</html>
```

AngularJS dispone de una gran cantidad de filtros predefinidos. En el bloque de código anterior, vemos cómo utilizamos, el filtro `uppercase` para transformar una cadena a mayúsculas una cadena, o el filtro `orderBy` nos permite ordenar una colección de elementos. También vemos que podemos encadenar tantos filtros como queramos. En la directiva `ng-repeat` usamos los filtros `filter` y `orderBy`.

Al encadenar filtros, el resultado se irá propagando al siguiente en el orden que los hemos declarado. En este caso, primero Hemos usado el filtro `filter` para filtrar elementos basándonos en el modelo `textFilter`. Luego, hemos ordenado el conjunto de resultados por la propiedad `name`.

Para no repetirnos en el uso de filtros ya tienen cierto coste computacional, podemos declarar variables en una directiva `ng-repeat`:

```
<ul>
 <li ng-repeat="person in filteredPeople = (people | filter:textFilter |
 orderBy:'name')">{{ person.name }} - {{ person.subject | uppercase }}</
li>
</ul>
```

```
Encontrados {{ filteredPeople.length }} resultados.
```

Nos detendremos en el capítulo dedicado a filtros a explicar cómo funciona cada uno de ellos. También, veremos cómo construir nuestros propios filtros.

2.5. Vistas, controladores y scope

Vamos a centrarnos ahora en la parte MVVM de AngularJS. Las Vistas y Controladores serán similares a lo que hemos visto en otros frameworks, y además disponemos de un elemento, llamado `scope`, que hace las veces de ViewModel.

En AngularJS, tenemos una vista, como las que hemos estado viendo en los anteriores ejemplos con sus filtros, sus directivas y su *data binding*. Como se ha comentado, no queremos declarar variables en la vista, porque hace nuestro código menos portable y testable. Para estos menesteres, disponemos de un objeto JavaScript llamado `Controller`, que va a gestionar qué datos se pasan a la vista, si éstos se actualizan y, también, va a comunicarse con el servidor en caso de que haya que actializar información en su lado.

Entre la vista y el controlador, hay un elemento llamado `scope`. El `scope` es el "pegamento" que une la vista y el controlador. Ni la vista sabe nada del controlador, ni el controlador de la vista. Nuevamente, esto nos permite hacer nuestro código muy modular y testable, al no haber una relación directa entre estos elementos.

Otra ventaja, es que podemos tener un único controlador vinculado a diferentes vistas. Por ejemplo, podemos tener una vista para una versión mobile otra para una versión desktop asociadas al mismo controlador.

¿Y qué es exactamente un ViewModel? Un ViewModel no es, ni más ni menos, que los datos que van a ser gestionados por la vista. Y eso es lo que el `scope` es.

Vamos a transformar nuestro último ejemplo, creando un controlador y pasando los datos a la vista, en lugar de declararlos en ella.

```
<!DOCTYPE html>
<html ng-app>
    <head>
        <title>Una tarde con AngularJS</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="stylesheet" href="components/lib/twitter-bootstrap/css/
bootstrap.css" />
    </head>
    <body>
        <div class="container" ng-controller="TeachersCtrl">
            <label>Filtro:</label> <input type="text" ng-
model="textFilter" /><br/>
            <ul>
                <li ng-repeat="person in filteredPeople = (people
 | filter:textFilter | orderBy:'name')">{{ person.name }} -
 {{ person.subject | uppercase }}</li>
            </ul>
            Encontrados {{ filteredPeople.length }} resultados.
        </div>
        <script src="https://code.angularjs.org/1.2.16/angular.min.js"></
script>
```

```
        <script>
            function TeachersCtrl($scope){
                $scope.people = [
                    {name:'Domingo', subject:'JPA'},
                    {name:'Otto', subject:'Backbone'},
                    {name:'Aitor',subject:'JavaScript'},
                    {name:'Miguel Ángel',subject:'JHD'},
                    {name:'Eli', subject:'REST'},
                    {name:'Fran', subject:'Grails'},
                    {name:'José Luís', subject:'PaaS'},
                    {name:'Alex', subject:'AngularJS'}
                ];
            }
        </script>
    </body>
</html>
```

Vemos que nuestro controlador es, simplemente, una función JavaScript. Una cosa interesante es que le pasamos como parámetro la variable `$scope`. Este `$scope` se pasa por Inyección de Dependencias, otra de las características de AngularJS de las que hemos hablado. En el momento en que se usa este controlador, AngularJS le inyectará de manera automática el objeto `$scope`. Una vez lo tenga, el controlador le añade una propiedad, llamada people, que es el array que anteriormente habíamos declarado en nuestra vista.

El controlador actúa como fuente de datos para la vista, pero no debería saber nada de la vista. Así que por eso inyectamos la variable `$scope`. Ésta va a permitir que nuestro controlador se comunique con la vista. El `$scope` pasará a la vista, una vez ésta sepa cuál es el controlador que la gestiona. Esto lo conseguimos gracias al atributo `ng-controller="TeachersCtrl"`. El `scope` estará visible para el `div` que lo ha llamado, así como para sus hijos. Vemos que el resto de nuestro código no ha variado y podemos seguir accediendo a la colección `people`. Solo que ahora estamos accediendo a la propiedad `people` del `$scope`.

2.6. Módulos, Rutas y Factorías

Ya hemos visto cómo funciona el *data binding*, las directivas, los filtros, controladores, cómo integramos una vista y un controlador gracias al objeto `$scope`... ¡Ya casi estamos listos para crear una aplicación completa en AngularJS!

Ahora vamos a ver cómo modularizar nuestra aplicación, y conceptos como Rutas y Factorías.

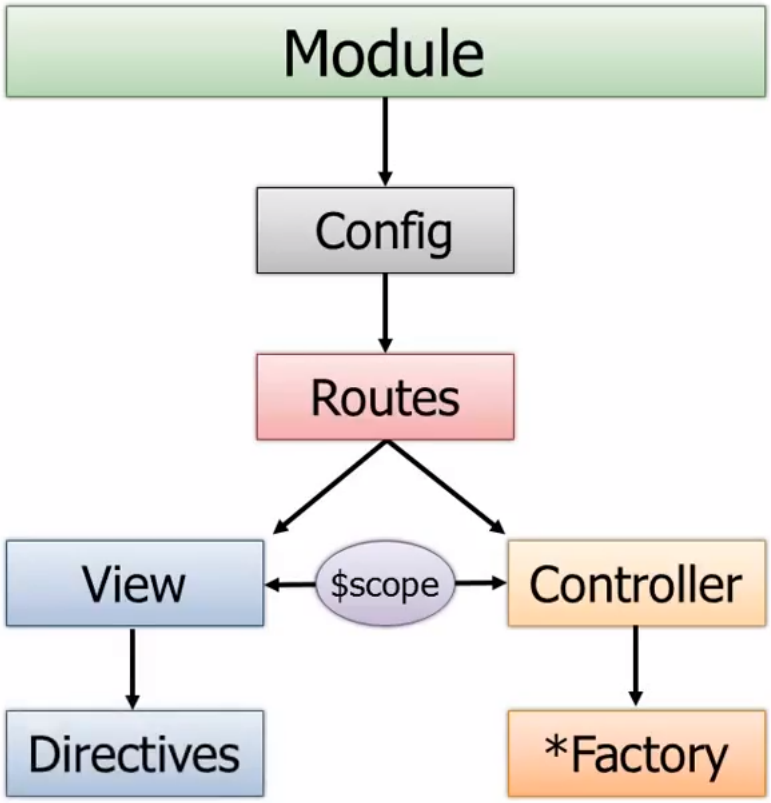
En AngularJS, un módulo puede tener una función de configuración. En esta función es donde, entre otras cosas, podemos definir rutas. Las rutas son importantes porque si nuestra aplicación tiene varias vistas, y éstas deben ser cargadas en nuestra página, debemos entonces tener un mecanismo para saber dónde estamos, qué vista tenemos asociada, qué controlador la gestiona…

Cuando definimos una ruta en AngularJS, también debemos definir dos elementos:

- Una vista. Por ejemplo, si estamos en la ruta `/profile`, entonces mostraremos la vista `/partials/profile.html`
- Un controlador. En lugar de definir el controlador en la vista con la etiqueta `ng-controller`, podemos establecerlo al definir una ruta. Siguiendo con el ejemplo anterior, asociaríamos el controlador `ProfileCtrl`

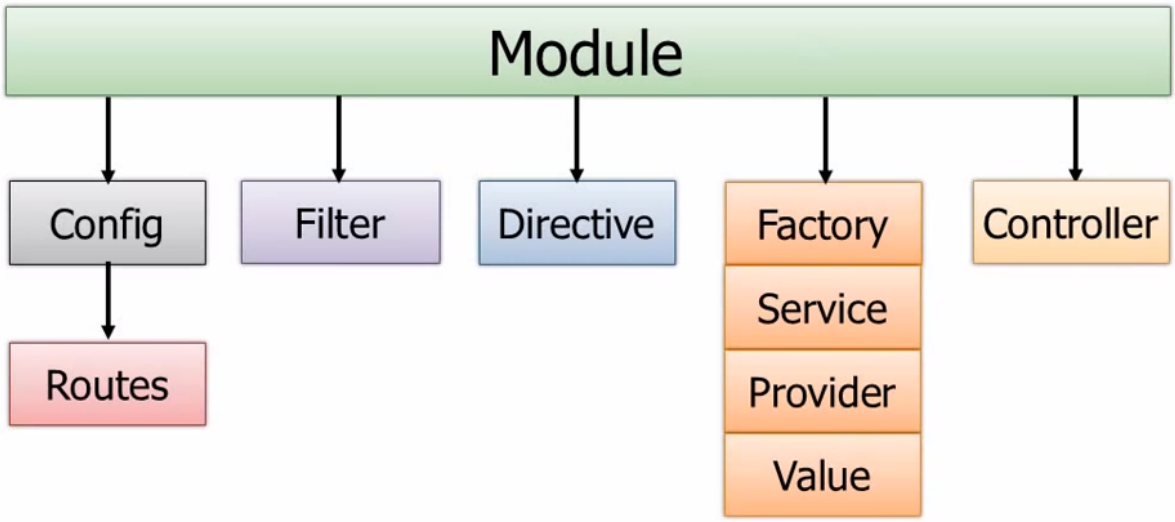
Un controlador no debería tener toda la lógica de la aplicación. En una aplicación bien estructurada, esta lógica la sacaríamos fuera a unos objetos que, en AngularJS, se llaman *Factories*, *Services* y *Providers*.

En la vista, tenemos el apoyo de las directivas y filtros, algunos de las cuales ya hemos visto.



Para poder definir rutas, tenemos que estructurar nuestra aplicación un poquito mejor de lo que hemos hecho hasta ahora. Aunque lo que hemos hecho hasta ahora es una aplicación AngularJS, no es la manera recomendada de implementarla.

En primer lugar, tenemos que definir un módulo en nuestra etiqueta `ng-app`, cosa que no habíamos hecho aún. Dentro de nuestro objeto `module` es donde vamos a poder configurar nuestras rutas, y también definir filtros, directivas, controladores, factorías, y demás servicios que serán específicos para nuestra app. Podríamos pensar en un `module` como un contenedor de objetos donde podemos tener todas estas cosas.



Lo importante es definir un nombre para nuestro módulo. Como estamos haciendo una pequeña aplicación que muestra un listado de profesores y asignaturas, la llamaremos `teachersApp`. Éste es el nombre que irá en nuestra etiqueta [ng-app].

```
<html ng-app="teachersApp">
```

Y ésta es la manera de crear un módulo en AngularJS:

```
var teachersApp = angular.module('teachersApp', []);
```

Fácil, ¿no?. Con esto, estamos creando un módulo, y diciendo a AngularJS que se va a llamar *teachersApp*. Vemos que está seguido por un array vacío. Aquí es donde vemos la potencia de la inyección de dependencias. Resulta que nuestro módulo puede incluir otros módulos de los que nuestra aplicación depende en cierto grado. Por ejemplo, en el siguiente fragmento estamos incluyendo un módulo llamado `myCustomModule`:

```
var teachersApp = angular.module('teachersApp', ['myCustomModule']);
```

Aquí, estamos diciendo a Angular que busque otro módulo, llamado `myCustomModule`, y lo inyecte en el nuestro, de manera que todos sus elementos (directivas, filtros, factorías, …) estarán disponibles en el momento que los inyectemos.

Haciendo una analogía, el incluir aquí estos elementos sería como hacer un `import` de un paquete completo en Java.

Crear un controlador para nuestro módulo no es nada distinto a como lo hemos hecho hasta ahora. La única diferencia radica en que hay que definirlo dentro de nuestro módulo:

```
teachersApp.controller('teachersCtrl', function($scope){
   $scope.people = [
        {name:'Domingo', subject:'JPA'},
        {name:'Otto', subject:'Backbone'},
```

```
        {name:'Aitor',subject:'JavaScript'},
        {name:'Miguel Ángel',subject:'JHD'},
        {name:'Eli', subject:'REST'},
        {name:'Fran', subject:'Grails'},
        {name:'José Luís', subject:'PaaS'},
        {name:'Alex', subject:'AngularJS'}
    ];
});
```

La manera de llamar al controlador desde la vista es exactamente la misma que hace un rato.

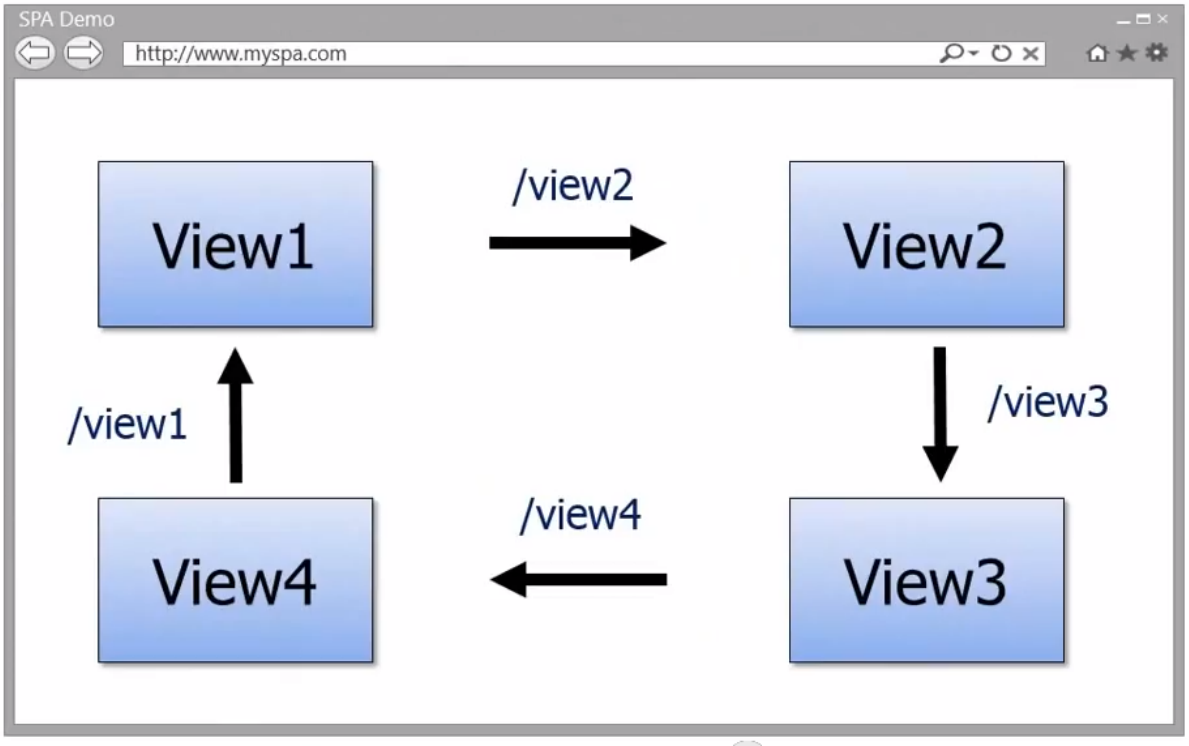
```
<!DOCTYPE html>
<html ng-app="teachersApp">
    <head>
        <title>Una tarde con AngularJS</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <link rel="stylesheet" href="components/lib/twitter-bootstrap/css/
bootstrap.css" />
    </head>
    <body ng-controller="TeachersCtrl">
        <div class="container">
            <label>Filtro:</label> <input type="text" ng-
model="textFilter" /><br/>
            <ul>
                <li ng-repeat="person in filteredPeople = (people
 | filter:textFilter | orderBy:'name')">{{ person.name }} -
 {{ person.subject | uppercase }}</li>
            </ul>
            Encontrados {{ filteredPeople.length }} resultados.
        </div>
        <script src="https://code.angularjs.org/1.2.16/angular.min.js"></
script>
        <script>
            var teachersApp = angular.module('teachersApp', []);

            teachersApp.controller('TeachersCtrl', function($scope){
               $scope.people = [
                    {name:'Domingo', subject:'JPA'},
                    {name:'Otto', subject:'Backbone'},
                    {name:'Aitor',subject:'JavaScript'},
                    {name:'Miguel Ángel',subject:'JHD'},
                    {name:'Eli', subject:'REST'},
                    {name:'Fran', subject:'Grails'},
                    {name:'José Luís', subject:'PaaS'},
                    {name:'Alex', subject:'AngularJS'}
                ];
            });
        </script>
    </body>
</html>
```

En el ejemplo, hemos decidido refactorizar nuestro código y crear una función anónima para el controlador. Recordemos que esto es JavaScript, donde las funciones se pueden pasar como argumentos. Otra manera de hacer lo mismo tocando menos código hubiera sido:

```
function TeachersCtrl($scope){
    $scope.people = [
        {name:'Domingo', subject:'JPA'},
        {name:'Otto', subject:'Backbone'},
        {name:'Aitor',subject:'JavaScript'},
        {name:'Miguel Ángel',subject:'JHD'},
        {name:'Eli', subject:'REST'},
        {name:'Fran', subject:'Grails'},
        {name:'José Luís', subject:'PaaS'},
        {name:'Alex', subject:'AngularJS'}
    ];
}
var teachersApp = angular.module('teachersApp', []);
teachersApp.controller('TeachersCtrl', TeachersCtrl);
```

Una vez hemos definido un módulo y un controlador, en algún momento vamos a tener que definir rutas para nuestra aplicación SPA.

Veamos aquí un ejemplo de rutas para una aplicación. En algún momento, debido a un evento, nuestra aplicación pasará de la vista 1 a la 2, mapeada por la ruta `/view2`, de la 2 a la 3… y así realizando un ciclo.



Lo importante aquí es saber que, con el cambio de ruta, no se recargará la página completa, sino el bloque que nosotros hayamos indicado.

La manera de definir estos bloques puede ser:

- Como una plantilla dentro de nuestro `index.html`.
- Como ficheros HTML independientes, habitualmente conocidos como `partials`. Ésta suele ser la manera habitual de hacerlo, sobre todo en aplicaciones grandes.

A partir de la versión 1.2, el módulo de rutas se extrajo del core de AngularJS como módulo independiente, con lo que lo primero que tendremos que hacer es traernos este módulo:

```
<script src="https://code.angularjs.org/1.2.16/angular-route.min.js"></
script>
```

Como hemos mencionado anteriormente, para incluir módulos extras en el nuestro, tenemos que declararlo en la definición:

```
var teachersApp = angular.module('teachersApp', ['ngRoute']);
```

Una vez hecho esto, ya podemos definir las rutas. Hemos dicho anteriormente que un módulo de AngularJS tenía una función de configuración, donde definíamos las rutas. Vamos a usar esta función, a la que tenemos que inyectarle un objeto llamado `$routeProvider`.

Vamos a definir dos rutas en nuestra aplicación:

- La primera será el listado de profesores que hemos hecho antes.
- La segunda consistirá en una página de asignaturas. Como aún no lo hemos implementado, mostrará un bonito mensaje de *En construcción*.

En cualquier otro caso, podemos definir una ruta por defecto a la que seremos redirigidos si introducimos algo a donde nuestra aplicación no sabe qué hacer con ello. Para este ejemplo, será la lista de profesores:

```
teachersApp.config(function($routeProvider){
    $routeProvider
        .when('/teachers',{
            controller : 'teachersCtrl',
            templateUrl : 'teachers.html'
        })
        .when('/subjects',{
            templateUrl : 'subjects.html'
        })
        .otherwise({ redirectTo : '/teachers'});
});
```

Para cada ruta, podemos definir una plantilla y un controlador. La configuración de rutas no obliga a introducir un par controlador - plantilla, como vemos en la segunda ruta.

Para adaptar nuestra aplicación a lo que acabamos de definir, tenremos que crearnos una página HTML, a la que llamaremos teachers.html. En ella estará el listado de profesores.

```
<label>Filtro:</label> <input type="text" ng-model="textFilter" /><br/>
<ul>
    <li ng-repeat="person in filteredPeople = (people | filter:textFilter
 | orderBy:'name')">{{ person.name }} - {{ person.subject | uppercase }}</
li>
</ul>
Encontrados {{ filteredPeople.length }} resultados.
```

Tendremos que refactorizar nuestro fichero `index.html` e introducir una nueva directiva, llamada `ng-view`.

```
<!DOCTYPE html>
<html ng-app="teachersApp">
    <head>
        ...
    </head>
    <body ng-controller="teachersCtrl">
        <div class="container">
            <ul class="nav nav-pills nav-justified">
                <li><a href="#/teachers">Profesores</a></li>
                <li><a href="#/subjects">Asignaturas</a></li>
            </ul>
            <ng-view></ng-view>
        </div>
        <script src="https://code.angularjs.org/1.2.16/angular.min.js"></
script>
        <script src="https://code.angularjs.org/1.2.16/angular-
route.min.js"></script>
        <script>
            ...
        </script>
    </body>
</html>
```

La directiva `ng-view` complementa al servicio `$route`, incluyendo la plantilla que hemos definido en la función `config`. Cada vez que la ruta cambia, la vista cambiará en función de lo configurado. La directiva `ng-view` puede ser tanto una etiqueta como un atributo, con lo que os la podréis encontrar habitualmente con la forma: `<div ng-view></div>`.

Justo antes de la directiva `ng-view` hemos definido dos links, que nos permitirán navegar por la aplicación. Al no estar incluídos dentro del bloque `ng-view`, éstos permanecerán invariables durante toda la navegación.

Haciendo click en el enlace *asignaturas*, llegaremos a una página en construcción, que mantiene los elementos. Observando la barra de dirección, veremos que la ruta ha cambiado. Y lo más increíble de todo, es que si hacemos click en el *back button* del navegador, volveremos al listado de profesores. El histórico del navegador funciona… ¡y sin hacer nada! AngularJS se encarga de gestionar el histórico del navegador.

Comunicándonos con el controlador

Ahora que tenemos nuestra lógica de navegación implementada, vamos a ver de qué manera podemos comunicarnos con el controlador. Para ello, en el listado de profesores, vamos a introducir la posibilidad de crear uno nuevo.

```
<h1>Profesorado</h1>
<label>Filtro:</label> <input type="text" ng-model="textFilter" /><br/>
<ul>
    <li ng-repeat="person in filteredPeople = (people | filter:textFilter
 | orderBy:'name')">{{ person.name }} - {{ person.subject | uppercase }}</
li>
</ul>
```

```
Encontrados {{ filteredPeople.length }} resultados.
<br/><br/>
<h2>Nuevo profesor</h2>
<label>Nombre:</label> <input type="text" ng-model="newTeacher.name" /
><br/>
<label>Asignatura</label> <input type="text" ng-model="newTeacher.subject"
 /><br/>
<button ng-click="addTeacher()">Guardar</button>
```

Fijémonos en el botón, que incorpora una nueva directiva, llamada `ng-click`. Ésta responde al evento `onClick`, realizando una llamada a la función `addTeacher()`.

Como hemos dicho antes, la vista no sabe nada del controlador. Pero disponemos del objeto `$scope`, que permite exponer elementos del controlador en la vista. Lo hemos hecho con una colección (el array de profesores), y también podemos exponer una función:

```
teachersApp.controller('teachersCtrl', function($scope){
    $scope.people = [
        ...
    ];

    $scope.addTeacher = function() {
        $scope.people.push({
            name : $scope.newTeacher.name,
            subject : $scope.newTeacher.subject
        });
    };
});
```

Vemos que no se ha pasado ningún dato como parámetro, ya que al pasarlo como `ng-model` en nuestros inputs, ya lo podemos obtener a través del `$scope`.

2.7. Usando factorías y servicios

Una de las cosas que observamos en nuestra aplicación, es que si pasamos del listado de profesores al de asignaturas y luego volvemos a los profesores otra vez, es que los que hayamos podido introducir han desaparecido. Esto es porque los datos están vinculados al objeto `$scope`, que se crea y se destruye con cada cambio de ruta.

AngularJS nos permite encapsular los datos de nuestra aplicación en una serie de elementos:

- Factorías
- Servicios
- Providers
- Valores
- Constantes

Los tres primeros (Factorías, Servicios y Providers), además de datos, nos permiten encapsular funcionalidades dentro de nuestra aplicación. Por ejemplo, si necesito mi lista de profesores en múltiples controladores, lo correcto sería guardarlos en uno de estos elementos.

Estos tres elementos pueden realizar la misma funcionalidad, y la diferencia entre ellos radica en cómo se crean. Lo veremos más adelante.

Además, estos elementos implementan el patrón singleton, lo que los convierte en los candidatos perfectos para intercambiar información entre controladores.

Así, ahora vamos a modificar nuestro código para utilizar una factoría, donde podremos obtener el listado de profesores, así como añadir ítems a la lista. Una factoría en AngularJS devuelve un objeto javascript, con lo que su forma será la siguiente:

```
teachersApp.factory('teachersFactory', function(){
    var teachers = [
        {name:'Domingo', subject:'JPA'},
        {name:'Otto', subject:'Backbone'},
        {name:'Aitor',subject:'JavaScript'},
        {name:'Miguel Ángel',subject:'JHD'},
        {name:'Eli', subject:'REST'},
        {name:'Fran', subject:'Grails'},
        {name:'José Luís', subject:'PaaS'},
        {name:'Alex', subject:'AngularJS'}
    ];

    return {
        getTeachers : function() {
            return teachers;
        },

        addTeacher : function(newTeacher) {
            teachers.push({
                name : newTeacher.name,
                subject : newTeacher.subject
            })
        }
    }
});
```

Más adelante en el curso, veremos de qué manera podemos obtener ese listado de profesores a través de una llamada AJAX o un servicio REST, en lugar de tener ese array *harcoded* en nuestra factoría.

Al crear la factoría, podemos usarla en nuestro controlador simplemente inyectándola como parámetro. Así, ya estará disponible y podremos usarla:

```
teachersApp.controller('teachersCtrl', function($scope, teachersFactory){
    $scope.people = teachersFactory.getTeachers();

    $scope.addTeacher = function() {
        teachersFactory.addTeacher($scope.newTeacher);
    };
});
```

La inyección de dependencias no se limita sólo a controladores, como veremos más adelante, podemos inyectar una factoría en otra factoría, o cualquiera de los elementos propios de AngularJS.

Como hemos dicho, una factoría implementa el patrón singleton. Esto significa que se instancia una única vez (la primera vez que es requerida), y está disponible durante toda la vida de la

aplicación. Ahora, si nos vamos al listado de asignaturas y volvemos al de profesores, veremos que no perdemos los datos que hayamos podido introducir.

Otra cosa que debemos saber, es que AngularJS permite encadenar operaciones. Esto significa que podemos refactorizar nuestro código de la siguiente manera:

```
angular
    .module('teachersApp', ['ngRoute'])
    .config(function($routeProvider){
        ...
    })
    .controller('teachersCtrl', function($scope, teachersFactory){
        ...
    })
    .factory('teachersFactory', function(){
        ...
    });
```

2.8. Ejercicio (1 punto)

Sube el código que hayas hecho en esta sesión, y aplícale el *tag* `intro` a la versión que quieres que se corrija.

Todo el código de la sesión estará dentro de una carpeta llamada `intro` también.

Tras haber realizado este breve tutorial, ya tenemos unos fundamentos bastante básicos del `core` de AngularJS. Opcionalmente, podemos ampliar nuestra aplicación realizando los siguientes incrementos:

- En el formulario de alta de profesores, introduciremos un nuevo campo que nos permita añadir la descripción de la asignatura.
- En el listado de asignaturas, mostraremos un listado de las asignaturas disponibles en el curso, y su descripción. Para ello crearemos el controlador subjectsCtrl
- Introduciremos la opción de eliminar un profesor. Quizá pueda ser interesante el uso de la función splice[3] de JavaScript, así como echarle un ojo a la documentación de la directiva `ngRepeat`[4] de AngularJS, a ver si podemos encontrar la manera de hacer uso de los índices.

[3] http://www.w3schools.com/jsref/jsref_splice.asp
[4] https://docs.angularjs.org/api/ng/directive/ngRepeat

3. Scopes

La mayoría de las aplicaciones web están basadas en el patrón MVC (Model-View-Controller). Sin embargo, MVC no es un patrón muy preciso, sino un patrón arquitectural de alto nivel. Además, existen muchas variaciones del patrón original, siendo los más conocidos MVP y MVVM. Para añadir un poco más de confusión, muchos frameworks y desarrolladores interpretan estos patrones de manera diferente. Esto da como resultado que el nombre MVC se use para describir diferentes arquitecturas y aproximaciones.

El equipo de AngularJS ha sido más pragmático con su aproximación, definiendo el framework como basado en el patrón MVW (Model-View-Whatever).

3.1. Hola mundo (otra vez)

Veamos nuevamente un típico ejemplo de *Hola mundo* para desgranar todos los elementos que intervienen:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hola mundo</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body ng-app="holaMundo">
    <div ng-controller="SaludaCtrl">
      Saluda a: <input type="text" ng-model="nombre" /><br/><br/>
      <h1>¡Hola, {{ nombre }}!</h1>
    </div>

    <script src="/angular.js/angular.js" type="text/javascript"></script>
    <script>
      angular
        .module('holaMundo', [])
        .controller('SaludaCtrl', function($scope){
          $scope.nombre = 'mundo';
        });
    </script>
  </body>
</html>
```

3.2. El objeto Scope

Siempre que queramos exponer un modelo a la vista (plantilla), haremos uso del objeto `$scope`. Para ello, simplemente deberemos asignar nuevas propiedades a una instancia de este objeto. Al hacerlo, ya estarán los valores disponibles en la plantilla.

Además, podemos también exponer funcionalidades a la vista, asociando funciones como propiedades de un `$scope`. Por ejemplo, podríamos crear un *getter* para la variable `nombre`:

```
<!DOCTYPE html>
<html>
```

```
  <head>
    <title>Hola mundo</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body ng-app="holaMundo">
    <div ng-controller="SaludaCtrl">
      Saluda a: <input type="text" ng-model="nombre" /><br/><br/>
      <h1>¡Hola, {{ getNombre() }}!</h1>
    </div>

    <script src="/angular.js/angular.js" type="text/javascript"></script>
    <script>
      angular
        .module('holaMundo', [])
        .controller('SaludaCtrl', function($scope){
          $scope.nombre = 'mundo';

          $scope.getNombre = function() {
            return $scope.nombre.toUpperCase();
          };
        });
    </script>
  </body>
</html>
```

De esta forma, podemos controlar precisamente qué parte del modelo y qué operaciones queremos exponer a la capa de presentación. Conceptualmente, un `$scope` tiene un comportamiento muy similar al de un *ViewModel* en el patrón MVVM.

Jerarquía y herencia de scopes

Cuando iniciamos una aplicación con AngularJS, se genera un scope a nivel de aplicación, llamado `$rootScope`. Desde ese momento, todo nuevo scope será hijo del `$rootScope`.

Podemos instanciar un nuevo `$scope` en cualquier momento a través del método `$new()`. Cuando lo instanciamos, un `$scope` hereda las propiedades de su padre, como vemos en este ejemplo:

```
var padre = $rootScope;
var hijo = padre.$new();

padre.saludo = "Hola";
hijo.nombre = "Mundo";

console.log(hijo.saludo); // --> 'Hola'

hijo.saludo = "Bienvenido";

console.log(hijo.saludo); // --> 'Bienvenido'
console.log(padre.saludo); // --> 'Hola'
```

Como hemos visto, podemos instanciar un nuevo `scope` en cualquier momento, lo normal es que AngularJS lo haga por nosotros cuando lo necesitemos. Por ejemplo, la directiva `ng-`

`controller` instancia un nuevo scope, que será el que inyecte en el controlador. En este caso, el scope será hijo del `$rootScope`.

A las directivas que crean nuevos *scopes* se las conoce como *scope creating directives*, y como hemos dicho será el propio AngularJS el encargado de crear los scopes por nosotros cuando se encuentre con una de estas directivas en el árbol del DOM.

> Los scopes forman una estructura de árbol, cuya raíz siempre será el `$rootScope`. Como la creación de scopes, está dirigida por el árbol del DOM, no debería resultar extraño que el árbol de scopes imite de alguna manera el árbol del DOM.

Ahora que sabemos que algunas directivas crean nuevos scopes hijos, quizá nos preguntemos por qué tanta complejidad. Para entenderlo, echemos un ojo a este ejemplo que utiliza la directiva `ng-repeat`:

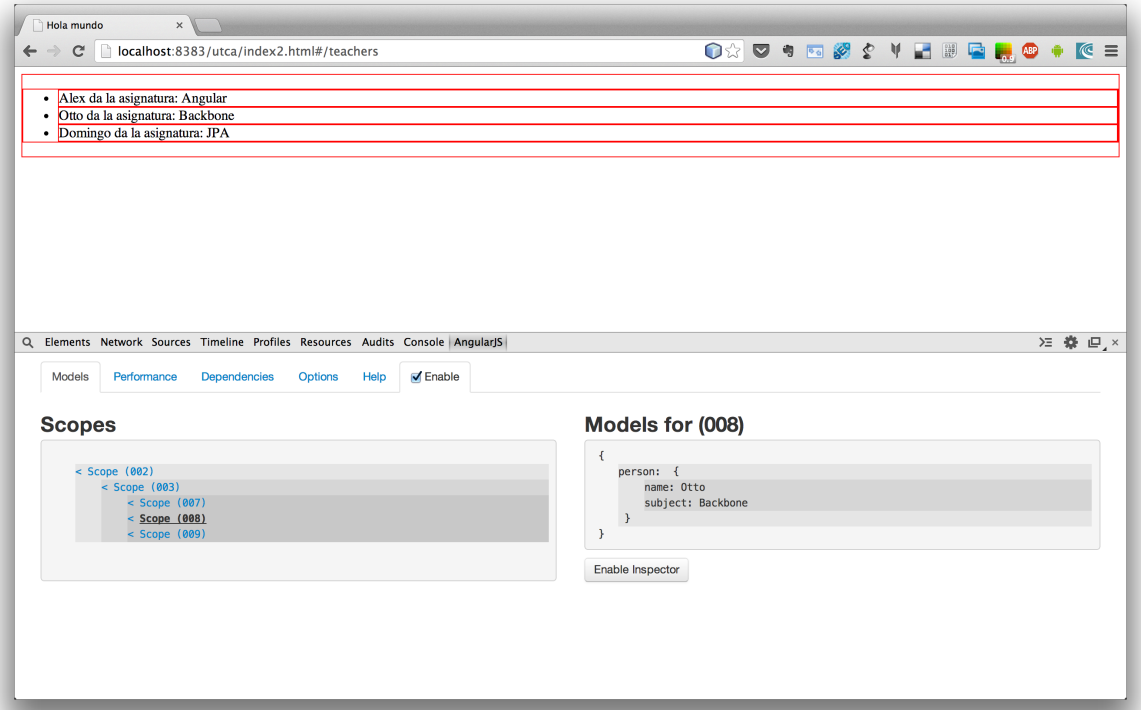
El controlador sería:

```
var peopleCtrl = function($scope){
  $scope.people = [
    { name:'Alex', subject: 'Angular', hours: 20},
    { name:'Otto', subject: 'Backbone', hours: 20},
    { name:'Domingo', subject: 'JPA', hours: 15}
  ];
};
```

Nuestra plantilla tendría la siguiente forma:

```
<ul ng-controller="peopleCtrl">
  <li ng-repeat="person in people">{{person.name}} da la asignatura:
 {{person.subject}} ({{ person.hours}} horas)</li>
</ul>
```

La directiva `ng-repeat` nos permite iterar sobre una colección de, en este caso, personas. Mientras itera, irá creando nuevos elementos en el DOM. El problema es que si utilizáramos el mismo `$scope`, estaríamos sobreescribiendo el valor de la variable `persona`. AngularJS soluciona este problema creando un nuevo `$scope` para cada elemento de la colección. Como se ha comentado, los scopes generan una jerarquía en forma de árbol, similar a la de los elementos del DOM. La extensión de AngularJS para Chrome nos permite verla:



Podemos ver en el pantallazo que cada *scope* (delimitado por un recuadro rojo) tiene su propio conjunto de valores del modelo. Así, cada item tiene su propio *namespace*, donde cada `<li>` posee un `scope` propio donde se puede definir la variable `person`.

Otra característica interesante de los objetos `scope` es que toda propiedad que definamos en un `scope` será visible para sus descendientes. Es muy interesante porque hace que no sea necesario redefinir elementos a medida que creamos scopes hijos.

Siguiendo con el ejemplo anterior, podemos calcular, en el `scope` del controlador padre, el número total de horas en base al conjunto de gente:

```
$scope.hours = $scope.people.reduce(function(value, person){
  return value ` person.hours;
}, 0);
```

En este caso, se ha hecho uso del método *reduce* de un Array.

Este método fue introducido en la versión 5 de ECMAScript[5], con lo que puede haber problemas de retrocompatibilidad con algunos navegadores. Aquí[6] podemos ver una tabla de compatibilidad de todas las funciones de ECMAScript 5 con los navegadores más usados del mercado.

La mayoría de estas funciones pueden implementarse en los navegadores que no las soportan de primeras, maximizando así la compatibilidad. Existen multiud de librerías que ya las implementan. Una de ellas es es5-shim[7].

5 http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
6 http://kangax.github.io/compat-table/es5/#Array.prototype.reduce
7 https://github.com/es-shims/es5-shim

> Siempre que se necesite hacer uso de estas funcionalidades y sea necesaria retrocompatibilidad, es muy recomendable usar librerías de este tipo.

Como habíamos comentado, este total de horas se propagará a los ámbitos hijos, que podrán hacer uso de él para, por ejemplo, determinar el porcentaje del total que supondrá cada asignatura:

```
<ul ng-controller="PeopleCtrl">
    <li ng-repeat="person in people">{{person.name}} da la asignatura:
 {{person.subject}} ({{ person.hours}} horas - {{person.hours / hours *
 100 | number:2}}%)</li>
</ul>
```

La herencia de scopes sigue el mismo patrón que la herencia prototípica de JavaScript: si no encontramos una propiedad en el objeto, subimos por el árbol de la jerarquía hasta dar con ella.

La herencia resulta muy sencilla de usar cuando estamos leyendo, pero sin embargo cuando estamos escribiendo puede darnos algún problema. Supongamos el siguiente bloque de código:

```
<div ng-app>
 <form name="myForm" ng-controller="Ctrl">
 Input dentro de un switch:
 <span ng-switch="show">
     <input ng-switch-when="true" type="text" ng-model="myModel" />
 </span>
     <br/>
 Input fuera del switch:
 <input type="text" ng-model="myModel" />
     <br/>
 Valor:
 {{myModel}}
 </form>
</div>
```

```
function Ctrl($scope) {
     $scope.show = true;
     $scope.myModel = 'hello';
}
```

Si manipulamos el segundo `input`, todos los elementos se modificarán a la vez. Pero, ¿qué sucede si modificamos el primero, y luego el segundo nuevamente? Parece como que el primero queda *desconectado* del resto. De hecho, se crea una nueva variable en el `scope` hijo que hace que esto funcione de esta manera. Podéis hacer la prueba usando el inspector de AngularJS para Chrome.

Esto se debe a la herencia prototípica (*prototipal inheritance* a partir de ahora) de JavaScript, y todas las reglas que se aplican a ésta, se aplican a los scopes, que al fin y al cabo son objetos JavaScript. El `scope` no es el modelo, sino que referencia al modelo. Por tanto, en el momento que modificamos el primer input, que tiene un `scope` propio, estamos referenciando a un nuevo modelo.

Esto es fácil de solucionar, haciendo uso de objetos. Podemos redeclarar el objeto `myModel` de la siguiente manera:

```
function Ctrl($scope) {
     $scope.show = true;
     $scope.myModel = { value: 'hello'};
}
```

Y usarlo así en nuestra vista:

```
<div ng-app>
 <form name="myForm" ng-controller="Ctrl">
 Input dentro de un switch:
 <span ng-switch="show">
     <input ng-switch-when="true" type="text" ng-model="myModel.value" />
 </span>
     <br/>
 Input fuera del switch:
 <input type="text" ng-model="myModel.value" />
     <br/>
 Valor:
 {{ myModel.value }}
 </form>
</div>
```

De esta manera, estaremos referenciando siempre al mismo objeto y no tendremos elementos desconectados sin querer.

Existe otra manera, que es hacer uso del elemento `parent`. Éste hace referencia al ámbito padre, y lo podríamos llamar de la siguiente manera en el *switch*:

```
<span ng-switch="show">
    <input ng-switch-when="true" type="text" ng-model="$parent.myModel" />
</span>
```

Sin embargo, esto resolvería nuestros problemas sólo si el dato estuviera en el ámbito padre, no si el padre también lo hubiera heredado. Además, no podemos estar seguros al 100% que `parent` va a ser el ámbito superior, ya que puede que estemos empleando alguna directiva que haya creado un `scope` adicional.

Podemos encontrar más información acerca de esto en este enlace[8]

También, en este vídeo[9], Miško Hevery hace una serie de reflexiones sobre cómo trabajar con los scopes.

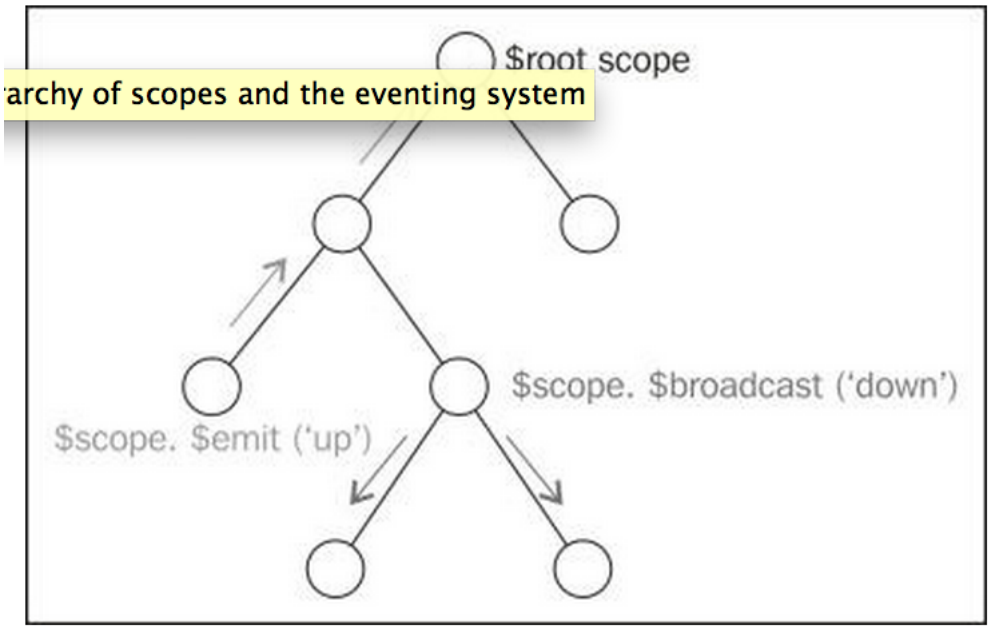
Propagación de eventos

Como hemos comentado, en nuestra aplicación se generará un árbol de objetos `scope` similar a la estructura del DOM. En la raíz de este árbol se encuentra el objeto `$rootScope`

[8] https://github.com/angular/angular.js/wiki/Understanding-Scopes
[9] https://www.youtube.com/watch?v=ZhfUv0spHCY&t=29m19s

Podemos usar esta jerarquía para transmitir eventos dentro del árbol, tanto en dirección ascendente con el método `scope.$emit` como descendente con `scope.$broadcast`.

La captura de estos eventos se realiza con el método `scope.$on`.



La función `scope.$emit(name, args);` envía un evento hacia arriba en la jearquía, notificando a todos los *listeners*. El ciclo de vida del evento comienza en aquel `scope` que ha llamado a `$emit`. Este evento irá hacia arriba hasta llegar al `$rootScope`, y todos los que hayan dado de alta un *listener* y estén en el camino del evento serán notificados. Los suscritos a un evento pueden cancelarlo.

Por su parte, la función `scope.$broadcast(name, args);` envía un evento hacia abajo en la jearquía, notificando todos los *listeners* herederos.

Ambos métodos tienen los mismos argumentos:

Param	Tipo	Detalles
`name`	`string`	Nombre del evento que se propaga.
`args`	`*`	Uno o más argumentos, que se propagarán con el evento.

Para suscribirnos a un evento, lo hacemos con `scope.$on(eventName, listener)`. Los parámetros son:

Param	Tipo	Detalles
`name`	`string`	Nombre del evento al que nos suscribimos
`listener`	`function(event, …args)`	Función a invocar cuando se recibe el evento.

El objeto `event` que se le pasa al listener tiene los siguientes atributos:

- `targetScope`: el `scope` en el que el evento fue emitido o difundido
- `currentScope`: el `scope` que maneja el evento.
- `name`: nombre del evento
- `stopPropagation`: esta función cancela el evneto y hace que no se siga propagando.
- `preventDefault`: establece el *flag* `defaultPrevented` a `true`.
- `defaultPrevented`: valdrá `true` si se ha llamado a la función `preventDefault`.

Eventos de AngularJS

Dentro del framework, existen tres eventos que se emiten

- `$includeContentRequested`
- `$includeContentLoaded`
- `$viewContentLoaded`

y siente eventos que se difunden

- `$locationChangeStart`
- `$locationChangeSuccess`
- `$routeUpdate`
- `$routeChangeStart`
- `$routeChangeSuccess`
- `$routeChangeError`
- `$destroy`

Podemos ver que se usan escasamemnte en el *core* de AngularJS. Pese a ser una manera sencilla de intercambiar datos entre controladores, debemos evaluar si es la mejor opción. Por ejemplo, en muchos casos puede ser útil el uso del *two-way data binding* para obtener una solución más sencilla.

Esto nos lleva a otro método interesante del objeto `scope`. Es el método `watch(watchExpression, [listener], [objectEquality])`, que registra un *listener* que se ejecuta cada vez que el resultado de la expresión *watchExpression* cambia.

La función `watchExpression` se llama en cada iteración del ciclo de vida de Angular, y debe devolver el valor que queramos observar.

El `listener` se ejecuta sólo cuando el valor de la `watchExpression` ha variado desde su última ejecución. Esta inecualidad se determina según la función angular.equals[10] También, se hace uso de la función angular.copy[11] para guardar el objeto, para utilizarlo en la siguiente

[10] https://docs.angularjs.org/api/ng/function/angular.equals
[11] https://docs.angularjs.org/api/ng/function/angular.copy

iteración de la comparación. La `watchExpression` debe ser lo más sencilla posible, ya que de otra manera podríamos tener problemas de rendimiento de rendimiento y memoria.

El `listener` puede modificar el modelo si así lo desea, lo que podría implicar la activación de otros *listeners*, re-lanzando los *watchers* hasta que no se detecta ningún cambio. Para prevenir entrar en bucles infinitos, existe un límite de re-lanzado de iteraciones, que es 10.

El siguiente ejemplo hace uso de una expresión y de una función de evaluación para observar una serie de cambios, y realizar una acción al respecto, que será contabilizar el número de cambios realizados sobre la variable.

```
<!DOCTYPE html>
<html ng-app="ses03.watch">
<head lang="en">
    <meta charset="UTF-8">
    <title></title>
</head>
<body ng-controller="WatchCtrl">
<p> <label>Nombre <input type="text" ng-model="name"/></label> </p>

<p> Contador de cambios en el nombre: {{ counter }} </p>

<p> <label>Alimento <input type="text" ng-model="food"/></label> </p>

<p> Contador de cambios en el alimento: {{ foodCounter }} </p>

<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.19/
angular.min.js"></script>
<script>
    angular
            .module('ses03.watch', [])
            .controller('WatchCtrl', function ($scope) {
                $scope.name = 'alex';
                $scope.counter = 0;

                //Usaremos una expresión para evaluar
                $scope.$watch('name', function (newValue, oldValue) {
                    $scope.counter++;
                });


                // Usaremos una función para evaluar
                $scope.food = 'paella';
                $scope.foodCounter = 0;
                $scope.$watch(
                        function () { return $scope.food.length; }, //
Función de evaluación

                        function (newValue, oldValue) { //Listener
                            if (newValue !== oldValue) {
                                $scope.foodCounter++;
                            }
                        }
                );
            });
</script>
</body>
```

```
</html>
```

El ciclo de vida del scope

El flujo normal de un browser cuando se recibe un evento es que éste termine llamando a una función JavaScript de *callback*. Una vez ésta ha finalizado, el browser renderiza el DOM de nuevo y espera a recibir más eventos.

Cuando se hace esta llamada JavaScript al navegador, el código se ejecuta fuera del contexto de ejecución de AngularJS, lo que significa que AngularJS no tiene ni idea de que se haya modificado el modelo. Para poder procesar estas modificaciones en el modelo, hay que hacer que todo lo que se ha hecho fuera del contexto de ejecución de AngularJS entre dentro de él a través del método `$apply`. Sólo las modificaciones del modelo que hagamos dentro de un método `$apply` serán tenidas en cuenta por AngularJS. Por ejemplo, una directiva como `ng-click` que escucha eventos del DOM, debe evaluar la expresión dentro de un método `$apply`. Así lo podemos ver En su código fuente[12].

Tras evaluar la expresión el método `$apply` realiza un `$digest`. En esta fase, el `scope` examina todas las expresiones `$watch` y compara sus resultados con los valores previos de manera asíncrona. Esto significa que una asignación como `$scope.username = 'admin'` no lanzará inmediatamente el listener de `$watch('username')`. En lugar de eso, se retrasará hasta la fase de `$digest`, de manera que se unifican las actualizaciones del modelo, y se garantiza que se ejecute una función `$watch` a la vez. Si un `$watch` cambia el modelo, forzará un ciclo `$digest` adicional.

Esto podemos resumirlo en las cinco fases por las que pasa una aplicación en AngularJS:

Creación

El `$injector` crea el objeto `rootScope` durante el *application bootstrap*. Cuando se produce el linkado de plantillas, algunas directivas crearán nuevos scopes.

Registro de watchers

Durante el linkado de plantillas, las directivas suelen registrar *watchers*. Éstos se usarán para propagar los valores del modelo al DOM.

Mutación del modelo

Para observar correctamente las mutaciones, habría que hacerlo dentro de `scope.$apply()`. Afortunadamente para nosotros, la API de AngularJS lo hace implícitamente, de manera que no es necesario hacerlo dentro de nuestros controladores si estamos realizando alguna tarea síncrona, o si estamos realizando tareas asíncronas con los servicios `$http`, `$timeout` o `$interval`.

Observación de la mutación

Al final de `$apply`, AngularJS realiza un ciclo `$digest` en el `rootScope` que se propagará posteriormente a todos los hijos. Durante este ciclo, todas las expresiones en un `$watch` se evaluarán para observar cambios en el modelo y, si esta se detecta, se invocará al *listener*.

[12] https://github.com/angular/angular.js/blob/master/src/ng/directive/ngEventDirs.js#L50

Destrucción

Cuando no se necesita más un `scope` hijo, su creador tiene la responsabilidad de destruirlo mediante una llamada a `scope.destroy()`. Esto detendrá la propagación llamadas `$digest` al hijo, y permitirá la llamada al recolector de basura para eliminar la memoria usada.

3.3. Ejercicios

Aplica el *tag* `scopes` a la versión que quieres que se corrija.

Calculadora (0,66 puntos)

Este ejercicio lo realizaremos en una carpeta llamada `calculator`.

Completa el siguiente código para implementar una calculadora que haga sumas, restas, multiplicaciones y divisiones.

```
<div ng-app>
  <h1>Calculadora</h1>

  <div ng-controller="CalcController">
    <div>
      <label>Primer operando <input type="number" /></label>
    </div>
    <div>
      <label>Segundo operando operando <input type="number" /></label>
    </div>
    <div>
      <button ng-click="">Suma</button>
      <button ng-click="">Resta</button>
      <button ng-click="">Multiplicación</button>
      <button ng-click="">División</button>
    </div>
    <h2>Resultado: XXX</h2>
  </div>
</div>
```

```
function CalcController($scope) {
  $scope.add = function() {

  };

  $scope.substract = function(a,b) {

  };

  $scope.divide = function() {

  };

  $scope.multiply = function(a,b) {

  };
```

```
}
```

Carrito de la compra (0,67 puntos)

Este ejercicio lo realizaremos en una carpeta llamada `shoppingcart`.

Vamos a hacer uso de la función `$watch` que nos ofrece el `scope` para implementar un sencillo carro de la compra.

Disponemos de una plantilla ya hecha, que muestra un listado de productos (generados aleatoriamente con JSON Generator[13]). También, disponemos de un controlador donde tenemos el listado de productos, una serie de variables y una función `addToCart` vacía.

Tendremos que implementar la función `addToCart`, para que añada ítems al carro. Además, implementaremos un *watcher* que observará cambios en el tamaño de dicho array. Cuando éstos se produzcan, actualizaremos la variable `$scope.totalItems` al número de ítems del carro. También, actualizaremos el valor de la variable `$scope.total`, con el importe total de los productos. Se recomienda hacer uso de la función `Array.prototype.reduce`[14] para calcular este total.

La plantilla de nuestro índice será:

```
<div ng-app>
  <div ng-controller="CartCtrl">
    <h2>{{totalItems}} ítems en la cesta ({{total}} &euro;)</h2>
    <div ng-repeat="product in products" style="float:left">
      <div>
        <img ng-src="{{product.picture}}" alt="" />
        <p>
          {{product.brand}} {{product.name}}
          <br/>
          {{product.price}} &euro;
        </p>
        <button ng-click="addToCart(product)">Añadir a la cesta</button>
      </div>
    </div>
  </div>
</div>
```

Por su parte, la plantilla de nuestro controlador será:

```
function CartCtrl($scope) {
  $scope.total = 0;
  $scope.totalItems = 0;
  $scope.cart = [];

  $scope.addToCart = function(product){
    //TODO: ADD TO CART
  };

  //TODO: WATCH
```

[13] http://beta.json-generator.com/OpatzFu
[14] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce

```
$scope.products = [
  {
      "_id": "54c688af49814edb036a2c33",
      "price": 145.4,
      "picture": "http://lorempixel.com/200/200/food/?id=846.9758",
      "brand": "adidas",
      "name": "Zone Job",
      "rating": 4
  },
  {
      "_id": "54c688afeffb1bad23bea9f7",
      "price": 137.24,
      "picture": "http://lorempixel.com/200/200/technics/?id=155.1389",
      "brand": "nike",
      "name": "Zoomair",
      "rating": 0
  },
  {
      "_id": "54c688af5f342fa2c06d2f3b",
      "price": 80.47,
      "picture": "http://lorempixel.com/200/200/food/?id=927.9926",
      "brand": "reebok",
      "name": "Volit",
      "rating": 2
  },
  {
      "_id": "54c688afbe7e2107363d0945",
      "price": 93.53,
      "picture": "http://lorempixel.com/200/200/animals/?id=387.7504",
      "brand": "nike",
      "name": "Konkis",
      "rating": 4
  },
  {
      "_id": "54c688af92223b7f877f096f",
      "price": 94.82,
      "picture": "http://lorempixel.com/200/200/nightlife/?id=296.9771",
      "brand": "adidas",
      "name": "Stockstring",
      "rating": 3
  },
  {
      "_id": "54c688af24de4b9fc39e0d48",
      "price": 109.24,
      "picture": "http://lorempixel.com/200/200/city/?id=427.4133",
      "brand": "adidas",
      "name": "Dong-Phase",
      "rating": 1
  },
  {
      "_id": "54c688afee99272b911e93fd",
      "price": 92.19,
      "picture": "http://lorempixel.com/200/200/nature/?id=580.5475",
      "brand": "adidas",
      "name": "Duozoozap",
      "rating": 0
  },
  {
```

```
        "_id": "54c688af3593d3f6a34bc2a4",
        "price": 82.37,
        "picture": "http://lorempixel.com/200/200/nightlife/?id=366.9091",
        "brand": "reebok",
        "name": "X-dom",
        "rating": 3
    },
    {
        "_id": "54c688af804d847b847935ac",
        "price": 76.53,
        "picture": "http://lorempixel.com/200/200/nature/?id=971.7978",
        "brand": "nike",
        "name": "Konkis",
        "rating": 3
    },
    {
        "_id": "54c688af96ba1759662c4274",
        "price": 90.01,
        "picture": "http://lorempixel.com/200/200/city/?id=37.581",
        "brand": "reebok",
        "name": "Ecooveit",
        "rating": 4
    },
    {
        "_id": "54c688afa4ee53c977c9ca3a",
        "price": 81.28,
        "picture": "http://lorempixel.com/200/200/transport/?id=752.8523",
        "brand": "adidas",
        "name": "Superstrong",
        "rating": 1
    },
    {
        "_id": "54c688af85e103fa79c83752",
        "price": 134.79,
        "picture": "http://lorempixel.com/200/200/fashion/?id=358.5133",
        "brand": "reebok",
        "name": "Superstrong",
        "rating": 5
    },
    {
        "_id": "54c688af04a986612841b7dc",
        "price": 76.37,
        "picture": "http://lorempixel.com/200/200/cats/?id=912.0469",
        "brand": "reebok",
        "name": "Fresh-Home",
        "rating": 0
    },
    {
        "_id": "54c688af5605253556078cff",
        "price": 147.47,
        "picture": "http://lorempixel.com/200/200/transport/?id=884.8266",
        "brand": "adidas",
        "name": "Touch-Hold",
        "rating": 2
    },
    {
        "_id": "54c688afa71c0978b878efd6",
        "price": 106.83,
```

```
        "picture": "http://lorempixel.com/200/200/fashion/?id=598.1251",
        "brand": "nike",
        "name": "Saorunlab",
        "rating": 2
    },
    {
        "_id": "54c688af0450426ca2d7680a",
        "price": 72.76,
        "picture": "http://lorempixel.com/200/200/food/?id=831.3375",
        "brand": "adidas",
        "name": "Konkis",
        "rating": 1
    },
    {
        "_id": "54c688afe30ff32f443e7c20",
        "price": 83.46,
        "picture": "http://lorempixel.com/200/200/sports/?id=604.4555",
        "brand": "adidas",
        "name": "Rank Sololax",
        "rating": 5
    },
    {
        "_id": "54c688afd24500b5cbc85148",
        "price": 77.19,
        "picture": "http://lorempixel.com/200/200/abstract/?id=333.8619",
        "brand": "reebok",
        "name": "Ecooveit",
        "rating": 0
    },
    {
        "_id": "54c688afb08a3950c86aa47f",
        "price": 82.05,
        "picture": "http://lorempixel.com/200/200/food/?id=118.5947",
        "brand": "nike",
        "name": "Zonedex",
        "rating": 5
    },
    {
        "_id": "54c688af234056a9e3f5c902",
        "price": 100.76,
        "picture": "http://lorempixel.com/200/200/technics/?id=20.7657",
        "brand": "reebok",
        "name": "Saorunlab",
        "rating": 0
    }
];
}
```

Ping-pong (0,67 puntos)

Este ejercicio lo realizaremos en una carpeta llamada `pingpong`.

En este ejercicio vamos a probar la propagación de eventos. Dispondremos de la siguiente plantilla HTML:

```
<div ng-app>
```

```
  <style>
    .ng-scope {
      border: 1px dotted red;
      margin: 5px;
    }
  </style>

  <div ng-controller="Controller1">
    <div ng-init="ping = 0"></div>
    <div ng-init="pong = 0"></div>

    <button ng-click="$emit('ping')">Emit event</button>
    <button ng-click="$broadcast('pong')">Broadcast event</button>

    <div>ping = {{ ping }}</div>
    <div>pong = {{ pong }}</div>

    <div ng-controller="Controller2">
      <button ng-click="">Emit event</button>
      <button ng-click="">Broadcast event</button>

      <div>ping = {{ ping }}</div>
      <div>pong = {{ pong }}</div>

      <div ng-controller="Controller2">
        <button ng-click="">Emit event</button>
        <button ng-click="">Broadcast event</button>

        <div>ping = {{ ping }}</div>
        <div>pong = {{ pong }}</div></div>

    </div>

  </div>
</div>
```

Y la siguiente plantilla JavaScript:

```
function Controller1($scope) {
  $scope.$on('ping', function(){
    $scope.ping``;
  });

  $scope.$on('pong', function(){
    $scope.pong``;
  });
}

function Controller2($scope) {

}

function Controller3($scope) {

}
```

Si ahora pulsamos en cualquiera de los dos primeros botones, veremos que los contadores *ping* y *pong* adquieren los mismos valores.

Deberemos:

- Emitir eventos *ping* y *pong* en el resto de botones.
- Introducir la lógica necesaria para que el evento que desencadene cada botón afecte al `scope` propio y a los ascendentes (en caso de `$emit`) o descendientes (en caso de `$broadcast`).

No se puede renombrar ninguno de los nombres de variable de la vista (todos deben llamarse `ping` y `pong`).

Tener en cuenta la herencia prototípica que hemos estado viendo.

Se ha introducido un poco de CSS para que se delimiten bien los tres `scope` que hay en la aplicación.

Si los bloques `ng-init` te molestan, puedes quitarlos <<<

4. Módulos y servicios

Desde el inicio de las sesiones, hemos visto que se han usado en varias ocasiones funciones globales para definir controladores. Esto acaba aquí, definirlo de esta manera es poco elegante, afecta a la estructura de la aplicación y hace el código más difícil de mantener y testear. Es por ello que AngularJS dispone de una serie de APIs que nos permiten definir módulos, y definir objetos dentro de ésto, de una manera muy sencilla.

4.1. Hola Mundo (y van tres)

Despidámonos para siempre de

```
var HelloCtrl = function ( $scope ) { $scope.name = 'World'; }
```

y digamos hola a

```
angular.module('helloApp', []) .controller ('HelloCtrl', function($scope)
{ $scope.name = 'World'; });
```

El objeto `angular` define una serie de utilidades. Una de ellas es module, que permite definir módulos. Un módulo es como una especie de contenedor de objetos gestionados por AngularJS, como por ejemplo los controladores.

4.2. Module

Para que el inyector de AngularJS sepa cómo crear y conectar todos estos objetos, neceista un registro de *recetas*. Cada *receta* tiene un identificador del objeto y la descripción de cómo crearlo.

Una *receta* debe pertenecer a un módulo de AngularJS, que es un saco que contiene una o más *recetas*. Y además, un módulo también puede contener información sobre otros módulos.

Cuando se inicia una aplicación de AngularJS con un módulo, AngularJS crea una instancia de un injector, que es quien crea el registro de las *recetas* como una unión de las existentes en el *core*, el módulo de la aplicación y sus dependencias El inyector consulta al registro de *recetas* cuando ve que tiene que crear un objeto para la aplicación.

Para definir un módulo, indicamos su nombre como primer argumento. El segundo argumento es un array, en el que incluiremos otros módulos, en caso de tener alguna dependencia. Ya vimos algo de esto en la sesión de introducción cuando introdujimos las rutas.

La llamada a `angular.module('helloApp', [])` devuelve una instancia de un módulo recién creado. Una vez tenemos la instancia, podemos definir controladores. Como hemos visto en el ejemplo, ésto se hace llamando a la función `controller(controllerName, controllerConstructor)`.

Una vez hemos definido un módulo, tenemos que informar a AngularJS de su existencia, cosa que haremos dando un valor a la directiva `ng-app`:

```
<body ng-app="helloApp">
```

La directiva `ngApp` designa el elemento raíz de nuestra aplicación, y habitualmente se coloca cerca del elemento raíz de la página, como las etiquetas `<body>` o `<html>`.

4.3. Servicios

Una vez hemos declarado un módulo, hemos dicho que podemos usarlo para registrar una serie de *recetas* para la creación de objetos de diversa índole.

Pero aunque sean de diversa índole, podríamos categorizar estos objetos en dos grandes grupos: **servicios** y **objetos especializados**.

Un servicio es un objeto cuya API está definida por el desarrollador que escribe dicho servicio.

Por su parte, un objeto especializado se ajusta a una API específica de AngularJS. Estos objetos pueden ser: controladores, directivas, filtros o animaciones.

El inyector de AngularJS necesita saber cómo crear estos objetos, y se lo decimos tipificando nuestro objeto a la hora de crearlo. Hay cinco tipos de *recetas*.

La más verbosa, pero también la más comprensible, es la del `Provider`. El resto (`Value`, `factory`, `Service` y `constant`) son sólo *azúcar sintáctico* sobre la definición de un `Provider`.

Veamos los diferentes escenarios para crear y usar servicios.

Todos los servicios en AngularJS son *singletons* [15]. Esto significa que el inyector usará las *recetas* como mucho una vez para crear el objeto. Posteriormente, éste se cacheará para la próxima vez que pueda hacer falta.

Como veremos, los creadores de AngularJS llamaron `Service` a una de estas *recetas* que designan servicios. De manera que cuando veamos su término en inglés nos estaremos refiriendo a la *receta* en

[15] http://en.wikipedia.org/wiki/Singleton_pattern

> concreto, mientras que cuando hagamos referencia al término *servicio* nos estaremos refiriendo a cualquiera de ellos.

Value

Digamos que queremos un servicio muy sencillo, llamado `clientId`, que nos devuelve un String que representa el identificador de usuario que usamos para alguna API remota. Lo podríamos definir de esta manera

```
var myApp = angular.module('myApp', []);
myApp.value('clientId', 'a123456le54321x');
```

Hemos creado un módulo de AngularJS llamado `myApp`. Posteriormente, hemoss dicho que este módulo contiene la *receta* para construir el servicio `clientId`, que en este caso únicamente devuelve una cadena.

Si quisiéramos mostrarlo vía *two-way data binding* lo haríamos de la siguiente manera:

```
myApp.controller('DemoController', ['clientId', '$scope', function
 DemoController(clientId, $scope) {
  $scope.clientId = clientId;
}]);
```

```
<html ng-app="myApp">
  <body ng-controller="DemoController">
    Client ID: {{clientId}}
  </body>
</html>
```

En este ejemplo, hemos usado la *receta* `Value` para dársela a `DemoCtrl` cuando invoca al servicio `clientId`.

Factory

Un `Value` es fácil de escribir, pero se echan de menos unos elementos importantes que necesitamos a menudo a la hora de escribir servicios. Así, pasaremos a conocer un elemento más complejo, llamado `factory`. Una `factory` nos permite:

- tener dependencias con otros servicios
- inicializar el servicio
- inicialización perezosa

Una `factory` construye un nuevo servicio mediante una función con cero o más argumentos. Estos argumentos son dependencias con otros servicios, que se inyectarán en tiempo de creación.

Una `factory` no es más que una versión más potente de un `Value`, de manera que podemos reescribir el servicio `clientId` de la siguiente manera:

```
myApp.factory('clientId', function clientIdFactory() {
```

```
  return 'a12345654321x';
});
```

Pero dado que el token no es más que un a cadena, quizá crear un `Value` sería más apropiado en este caso, y el código sería además más sencillo de interpretar.

Digamos que, por ejemplo, queremos crear un servicio encargado de calcular un *token* para autenticarse contra una API remota. Este *token* se llamará `apiToken` y se calculará en función del valor de `clientId`, y de una clave secreta guardada en el almacenamiento local del navegador:

```
myApp.factory('apiToken', ['clientId', function apiTokenFactory(clientId)
 {
  var encrypt = function(data1, data2) {
    // NSA-proof encryption algorithm:
    return (data1 + ':' + data2).toUpperCase();
  };

  var secret = window.localStorage.getItem('myApp.secret');
  var apiToken = encrypt(clientId, secret);

  return apiToken;
}]);
```

En el código anterior, vemos cómo se define el servicio `apiToken` con la *receta* de una `factory` que depende del servicio `clientId`. Entonces crea un token de autenticación a través de una encriptación hiperpotente indescifrable por la NSA [16].

Entre las *best practices* se recomienda nombrar una `factory` de la forma `<nombreDelServicio>Factory`. Aunque nadie lo requiere, ayuda a la hora de revisar código, o bien a la hora de *debuggear*.

De igual manera que un `Value` una `factory` puede crear un servicio de cualquier tipo, ya sea una primitiva, un objeto, una función o una instancia de un tipo de dato propio. primitive, object literal, function, or even an instance of a custom type.

Service

Los desarrolladores JavaScript usan tipos de datos *custom* para escribir código OO. Veamos cómo podríamos lanzar un Unicornio [17] al espacio mediante un servicio unicornLauncher, que es una instancia del siguiente objeto:

```
function UnicornLauncher(apiToken) {

  this.launchedCount = 0;
  this.launch = function() {
    // make a request to the remote api and include the apiToken
    ...
    this.launchedCount++;
  }
```

[16] http://en.wikipedia.org/wiki/National_Security_Agency
[17] http://en.wikipedia.org/wiki/Unicorn

```
}
```

Ya podemos lanzar unicornios al espacio, pero démonos cuenta que nuestro lanzador requiere un `apiToken`. Lo bueno es que ya habíamos creado una `factory` que resolvía este problema.

```
myApp.factory('unicornLauncher', ["apiToken", function(apiToken) {
  return new UnicornLauncher(apiToken);
}]);
```

Éste es, precisamente, el caso de uso más adecuado para un `Service`.

La *receta* de un `Service` produce un servicio, de igual manera que habíamos visto con `Value` y `factory`, pero lo hace invocando un constructor mediante el operador new. El constructor puede recibir cero o más argumentos, que representan dependencias que necesita la instancia de este tipo.

Además, un `Service` sigue un patrón de diseño llamado *constructor injection*. De manera que es el propio AngularJS quien se encarga de instanciar un nuevo objeto de la clase dada. Como nuestro UnicornLauncher tiene un constricutor, podemos reemplazar la `factory` por un `Service`, de la siguiente manera:

```
myApp.service('unicornLauncher', ["apiToken", UnicornLauncher]);
```

Provider

Como se ha dicho anteriormente, el `Provider` es la base sobre la que se crean el resto de servicios que hemos visto en los apartados anteriores. Precisamente porque es la base sobre la que se asientan el resto, no será difícil comprender que es la que nos ofrece mayores posibilidades. Pero en la mayoría de los casos todo lo que ofrece es excesivo, y será recomendable hacer uso de los otros servicios.

La receta de un `Provider` se define como un tipo propio que implementa un método `$get`. Este método es una función factoría [18], como el que se usa en una `factory`. De hecho, si definimos una `factory`, lo que se hace es crear un `Provider` vacío cuyo método `$get` apunta directamente a nuestra función.

Deberíamos usar un `Provider` cuando queremos introducir cierta configuración que esté disponible a para toda la aplicación. Para asegurar esto, esto debe hacerse antes de que se ejecute la aplicación, en una fase llamada *fase de configuración*. De esta manera, podemos crear servicios reutilizables cuyo comportamiento podría cambiar ligeramente entre aplicaciones.

Por ejemplo, nuestro lanzador de unicornios es tan potente y útil que lo van a usar muchas de las aplicaciones del parque de aplicaciones de nuestra empresa. Por defecto, el lanzador de unicornios los proyecta al espacio sin ningún tipo de protección ni escudo. Pero en algunos planetas, la atmósfera es tan pesada que debemos proteger a nuestros unicornios con papel de plata para evitar que se incineren al atravesar la atmósfera y así evitar su extinción. Estaría muy bien que pudiéramos configurar esto nuestro lanzador, y usarlo en la app que haga falta. Lo haríamos configurable de la siguiente manera:

[18] http://en.wikipedia.org/wiki/Factory_method_pattern

```
myApp.provider('unicornLauncher', function UnicornLauncherProvider() {
  var useTinfoilShielding = false;

  this.useTinfoilShielding = function(value) {
    useTinfoilShielding = !!value;
  };

  this.$get = ["apiToken", function unicornLauncherFactory(apiToken) {

    // let's assume that the UnicornLauncher constructor was also changed
 to
    // accept and use the useTinfoilShielding argument
    return new UnicornLauncher(apiToken, useTinfoilShielding);
  }];
});
```

Para activar el traje espacial de papel de plata, necesitamos crear una función `config` en la API de nuestro módulo, e inyectar en ella el `unicornLauncherProvider`:

```
myApp.config(["unicornLauncherProvider", function(unicornLauncherProvider)
 {
  unicornLauncherProvider.useTinfoilShielding(true);
}]);
```

Observemos que el `Provider` ha sido inyectado en la función de configuración. Esta inyección la hace in *provider injector*, que es distinto del inyector de instancias habitual que usaremos en el resto de nuestra aplicación. Este inyector únicamente trabaja con *providers*, ya que el resto de objetos no se crean en la fase de configuración.

Durante la inicialización de la aplicación, antes de que AngularJS haya creado ningún servicio, configura e instancia los *providers*. A esto lo llamamos la **fase de configuración** del ciclo de vida de la aplicación. Durante esta fase, como hemos dicho, los servicios no son accesibles porque aún no se han creado.

Una vez se ha finalizado la fase de configuración, ya no se puede interactuar con un `Provider` y empieza el proceso de crear servicios. A esta parte del ciclo de vida de la aplicación se le conoce como **fase de ejecución**.

Constant

Hemos visto cómo AngularJS divide el ciclo de vida de una aplicación en las fases de configuración y ejecución, y cómo se puede dotar de configuración a la aplicación a través de la función `config`. Dado que la función `config` se ejecuta en una fase en la que no tenemos servicios disponibles, no se tiene acceso ni siquiera a objetos sencillos creados con la utilidad `Value`.

Sin embargo, podemos tener valores tan simples como un prefijo de una url, que no necesiten dependencias o configuración, y que sean útiles en las fases de configuración y ejecución. Para esto sirve la utilidad `constant`.

Supongamos que nuestro servicio unicornLauncher puede estampar, en un unicorion, el nombre del planeta contra el que está siendo lanzado en la fase de configuración. El nombre

del planeta es específico para cada aplicación, y también lo usan muchos controladores en tiempo de ejecución. Podemos definir entonces el nombre del planeta como una constante:

```
myApp.constant('planetName', 'Greasy Giant');
```

Ahora, podemos configurar nuestro `unicornLauncherProvider` de la siguiente manera:

```
myApp.config(['unicornLauncherProvider', 'planetName', function(unicornLauncherProvider
 planetName) {
  unicornLauncherProvider.useTinfoilShielding(true);
  unicornLauncherProvider.stampText(planetName);
}]);
```

Y dado que una `constant` hace que el valor también esté disponible en la fase de ejecución, podemos usarla también en nuestros controladores:

```
myApp.controller('DemoController', ["clientId", "planetName", function
 DemoController(clientId, planetName) {
  this.clientId = clientId;
  this.planetName = planetName;
}]);
```

```
<html ng-app="myApp">
  <body ng-controller="DemoController as demo">
   Client ID: {{demo.clientId}}
   <br>
   Planet Name: {{demo.planetName}}
  </body>
</html>
```

4.4. Objetos de propósito especial

Anteriormente, hemos mencionado que tenemos objetos de propósito especial, cuya funcionalidad es diferente de la que ofrece un servicio. Estos objetos extienden el framework como *plugins*, implementando interfaces definidas por AngularJs. Estas interfaces son: `controller`, `directive`, `filter` y `animation`.

A excepción del objeto `controller`, el inyector usa la *receta* de una `factory` para crear estos objetos.

Ya hemos visto algo de estos objetos en las sesiones anteriores, y profundizaremos más en los siguietnes capítulos.

4.5. En resumen

Un inyector usa una serie de *recetas* para crear dos tipos de objetos: servicios y objetos de propósito especial. Para crear servicios, usamos cinco tipos de *receta* dinstintos: `Value`, `factory`, `Service`, `Provider` y `constant`.

De ellos, los más comunes són `factory` y `Service`, y sólo se distinguen en que los `Service` funcionan mejor con tipos de objetos ya definidos, y una `factory` devuelve

funciones y primitivas JavaScript. Un `Provider` es la receta "padre" de todos ellos, que no son más que *azúcar sintáctico* de un `Provider`. Un `Provider` es la *receta* más potente, pero no es necesaria a menos que necesitemos un componente reutilizable que requiera de algún tipo de configuración a nivel de aplicación, y es por esto que es el único elemento disponible en la fase de configuración de un aplicación.

4.6. Ejercicios

Aplica el *tag* `modules` a la versión que quieres que se corrija.

Convertir a módulo (0,66 puntos)

Vamos a coger el carrito de la compra del ejercicio anterior, y crear un módulo llamado `org.expertojava.carrito`, que no tenga ninguna dependencia.

El controlador deberá pasar ahora a formar parte del módulo.

Creación de una factoría (0,67 puntos)

En nuestro módulo, debemos crear una factoría. La llamaremos `productsFactory` y expondrá un único método, llamado `getProducts()`, que devolverá el listado de productos que teníamos en el controlador. Inyectaremos la factoría en el controlador, y ahora el listado de productos será `$scope.products = productsFactory.getProducts()`.

Creación de un servicio (0,67 puntos)

En nuestro módulo, también crearemos un servicio al que llamaremos `shoppingCartService`. Dicho servicio tendrá tres métodos: `addToCart(product)`, `getCart()` y `getTotal()`. Inyectaremos el servicio en nuestro controlador, donde refactorizaremos nuestra lógica: `$scope.addToCart = shoppingCartService.addToCart`, `$scope.total = shoppingCartService.getTotal()` y `$scope.cart = shoppingCartService.getCart()`. <<<

5. Filtros

Un filtro se encarga de formater el valor de una expresión, para ofrecérsela al usuario sin modificar el valor original. Puede usarse en vistas, controladores o servicios y son muy sencillos de declarar y programar.

La API subyacente es el `filterProvider`.

5.1. Cómo usar un filtro

Un mismo filtro se puede utilizar de dos maneras diferentes, en función de si lo hacemos desde una vista o desde código JavaScript.

En una vista

Para usar un filtro en una vista, podemos aplicar una expresión con la siguiente sintaxis:

```
{{ expression | filter }}
{{ 12 | currency }} <!-- 12€ -->
```

Los filtros, además, pueden encadenarse. En el siguiente ejemplo, la salida del primer filtro se pasa como entrada del segundo.

```
{{ expression | filter1 | filter2 }}
{{ 12 | number:2 | currency }} <!-- 12.00€ -->
```

En controladores, servicios y directivas

Para usar un filtro en un controlador/servicio/directiva, debemos inyectar una dependencia con `<nombreDelFiltro>Filter`. Por ejemplo, para usar el filtro `number` que hemos visto en el ejemplo anterior (que formatea un número con los decimales indicados en el segundo parámetro), inyectaríamos `numberFilter`.

```
angular.module('FilterInControllerModule', []).
  controller('FilterController',
 ['$scope', 'numberFilter', function($scope, numberFilter) {
    $scope.filteredText = numberFilter(12,2);
  }]);
```

Otra manera consiste en inyectar el servicio `$filter` en nuestro código javascript. Con él, podemos llamar a todos los filtros de la siguiente manera:

```
angular.module('FilterInControllerModuleV2', []).
  controller('FilterController', ['$scope', '$filter', function($scope,
 $filter) {
    $scope.filteredText = $filter('number')(12,2);
  }]);
```

5.2. Filtros predefinidos en AngularJS

filter

El filtro `filter`[19] selecciona un subset de ítems dentro de un array, devolviéndolo en un nuevo array

En una plantilla HTML lo usaremos de la forma:

```
{{ filter_expression | filter : expression : comparator}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('filter')(array, expression, comparator)
```

El parámetro `array` se corresponde con el array a filtrar. Por su parte, `expression` puede ser una cadena (búsqueda en todos los objetos), o bien un objeto (que servirá de

[19] https://docs.angularjs.org/api/ng/filter/filter

"ejemplo" para hacer las búsquedas en el array). El último parámetro es opcional nos permite implementar una función de comparación customizada.

Ejemplo:

```
<div ng-app>
  <div ng-init="friends = [
    {name:'John', phone:'555-1276'},
    {name:'Mary', phone:'800-BIG-MARY'},
    {name:'Mike', phone:'555-4321'},
    {name:'Adam', phone:'555-5678'},
    {name:'Julie', phone:'555-8765'},
    {name:'Juliette', phone:'555-5678'}
  ]"></div>

  Search: <input ng-model="searchText">
  <table id="searchTextResults">
    <tr><th>Name</th><th>Phone</th></tr>
    <tr ng-repeat="friend in friends | filter:searchText">
      <td>{{friend.name}}</td>
      <td>{{friend.phone}}</td>
    </tr>
  </table>
  <hr>
  Any: <input ng-model="search.$"> <br>
  Name only <input ng-model="search.name"><br>
  Phone only <input ng-model="search.phone"><br>
  Equality <input type="checkbox" ng-model="strict"><br>
  <table id="searchObjResults">
    <tr><th>Name</th><th>Phone</th></tr>
    <tr ng-repeat="friendObj in friends | filter:search:strict">
      <td>{{friendObj.name}}</td>
      <td>{{friendObj.phone}}</td>
    </tr>
  </table>
</div>
```

currency

El filtro `currency`[20] nos permite expresar un número en formato moneda. En una plantilla HTML lo usaremos de la forma:

```
{{ currency_expression | currency : symbol : fractionSize}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('currency')(amount, symbol, fractionSize)
```

Tanto `currency` como `fractionSize` son opcionales. Si no ponemos nada, se expresará en dólares con el formato de separación de miles y decimales estadounidense. Para expresarlo

[20] https://docs.angularjs.org/api/ng/filter/currency

a una región en concreto, lo mejor es instalar el módulo `ngLocale`[21] que corresponda, en nuestro caso sería `angular-locale_es-es.js`. Según la versión de AngularJS que estemos utilizando, tendremos que añadirlo a nuestro módulo principal.

```
angular.module('myModule', ['ngLocale'])
```

Ejemplo:

```
<script src="http://path/to/angular-locale_es-es.js"></script>
<div ng-app>
  {{ 1234567890.25 | currency }}
</div>
```

- Ejemplo de moneda con locale por defecto de AngularJs[22]
- Ejemplo de moneda con locale español[23]

date

El filtro `date`[24] nos permite formatear una fecha de la manera deseada. En una plantilla HTML lo usaremos de la forma:

```
{{ date_expression | date : format : timezone}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('date')(date, format, timezone)
```

El parámetro `date` puede ser de varios tipos, aunque lo más habitual es que sea un objeto `Date` o una fecha en milisegundos. Tanto `format` como `timezone` son opcionales. En la referencia de este filtro en la web de AngularJS podemos ver todas las formas que acepta.

El filtro `date` también se ve afectado por el módulo `ngLocale`. Ejemplo:

```
<script src="http://path/to/angular-locale_es-es.js"></script>
<div ng-app>
  <span ng-non-bindable>{{1288323623006 | date:'medium'}}</span>:
  <span>{{1288323623006 | date:'medium'}}</span><br>
  <span ng-non-bindable>{{1288323623006 | date:'yyyy-MM-dd HH:mm:ss Z'}}</
span>:
  <span>{{1288323623006 | date:'yyyy-MM-dd HH:mm:ss Z'}}</span><br>
  <span ng-non-bindable>{{1288323623006 | date:'MM/dd/yyyy @ h:mma'}}</
span>:
  <span>{{'1288323623006' | date:'MM/dd/yyyy @ h:mma'}}</span><br>
  <span ng-non-bindable>{{1288323623006 | date:"MM/dd/yyyy 'at' h:mma"}}</
span>:
  <span>{{'1288323623006' | date:"MM/dd/yyyy 'at' h:mma"}}</span><br>
```

21 https://github.com/angular/angular.js/tree/master/src/ngLocale
22 http://codepen.io/alexsuch/pen/EaXWyK?editors=101
23 http://codepen.io/alexsuch/pen/gbWgZe?editors=101
24 https://docs.angularjs.org/api/ng/filter/date

```
</div>
```

- Ejemplo de `date` con locale español[25]
- Ejemplo de `date` con locale por defecto[26]

json

El filtro `json`[27] recibe un objeto como entrada, y devuelve una cadena representando dicho objeto en formato JSON. En nuestro código HTML se usa de la siguiente forma:

```
{{ json_expression | json : spacing}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('json')(object, spacing)
```

El parámetro `spacing` es opcional, e indica el número de espacios que se utilizará en la indetación (el valor por defecto es 2). En el este enlace[28] podemos ver el siguiente ejemplo funcionando:

```
<div ng-app>
  <h3>Default spacing</h3>
  <pre id="default-spacing">{{ {nombre:'Alejandro', asignatura:'Frameworks
 JavaScript: AngularJS'} | json }}</pre>
  <h3>Custom spacing</h3>
<pre id="custom-spacing">{{ {nombre:'Domingo', asignatura: 'Frameworks de
 persistencia: JPA'} | json:4 }}</pre>
</div>
```

limitTo

El filtro `limitTo`[29] recibe como entrada un array, del que tomará un número de elementos igual al parámetro recibido (`limit`). Estos elementos se tomaran del principio si el número es positivo, y del final si es negativo. En nuestro código HTML se usa de la siguiente forma:

```
{{ limitTo_expression | limitTo : limit}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('limitTo')(input, limit)
```

Ejemplo[30]:

[25] http://codepen.io/alexsuch/pen/LELWRW?editors=101
[26] http://codepen.io/alexsuch/pen/wBeJzY?editors=101
[27] https://docs.angularjs.org/api/ng/filter/json
[28] http://codepen.io/alexsuch/pen/pvweNM
[29] https://docs.angularjs.org/api/ng/filter/limitTo
[30] http://codepen.io/alexsuch/pen/Pwjpmd

```
<script>
  angular.module('limitToExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.numbers = [1,2,3,4,5,6,7,8,9];
      $scope.letters = "abcdefghi";
      $scope.longNumber = 2345432342;
      $scope.numLimit = 3;
      $scope.letterLimit = 3;
      $scope.longNumberLimit = 3;
    }]);
</script>
<div ng-app="limitToExample">
  <div ng-controller="ExampleController">
    Limit {{numbers}} to: <input type="number" step="1" ng-
model="numLimit">
    <p>Output numbers: {{ numbers | limitTo:numLimit }}</p>
    Limit {{letters}} to: <input type="number" step="1" ng-
model="letterLimit">
    <p>Output letters: {{ letters | limitTo:letterLimit }}</p>
    Limit {{longNumber}} to: <input type="number" step="1" ng-
model="longNumberLimit">
    <p>Output long number: {{ longNumber | limitTo:longNumberLimit }}</p>
  </div>
</div>
```

lowercase

El filtro `lowercase`[31] convierte una cadena a minúsculas. En nuestro código HTML se usa de la siguiente forma:

```
{{ lowercase_expression | lowercase}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('lowercase')(lowercase_expression)
```

Ejemplo:

```
{{ 'HOLA MUNDO' | lowercase }}
```

uppercase

El filtro `uppercase`[32] convierte una cadena a mayúsculas. En nuestro código HTML se usa de la siguiente forma:

```
{{ uppercase_expression | uppercase}}
```

[31] https://docs.angularjs.org/api/ng/filter/lowercase
[32] https://docs.angularjs.org/api/ng/filter/uppercase

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('uppercase')(uppercase_expression)
```

Ejemplo:

```
{{ 'hola mundo' | uppercase }}
```

number

El filtro `number`[33] formatea un número en el locale que hayamos importado, y con el número de decimales indicado en su segundo parámetro, que es opcional.

```
En nuestro código HTML se usa de la siguiente forma:
```

```
{{ number_expression | number : fractionSize}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('number')(number, fractionSize)
```

Ejemplo[34]:

```
<script>
angular.module('numberFilterExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.val = 1234.56789;
    }]);
</script>
<div ng-app="numberFilterExample">
<div ng-controller="ExampleController">
  Enter number: <input ng-model='val'><br>
  Default formatting: <span id='number-default'>{{val | number}}</
span><br>
  No fractions: <span>{{val | number:0}}</span><br>
  Negative number: <span>{{-val | number:4}}</span>
</div>
<div>
```

orderBy

El filtro `orderBy`[35] nos permite ordenar un array por una propiedad, en sentido normal o inverso. En nuestro código HTML se usa de la siguiente forma:

[33] https://docs.angularjs.org/api/ng/filter/number
[34] http://codepen.io/alexsuch/pen/XJgMad
[35] https://docs.angularjs.org/api/ng/filter/orderBy

```
{{ orderBy_expression | orderBy : expression : reverse}}
```

En nuestro código javascript lo usaremos de la siguiente forma:

```
$filter('orderBy')(array, expression, reverse)
```

Veamos un ejemplo[36] de cómo funciona, y lo sencillo que es el uso de `expression`:

```
<script>
  angular.module('orderByExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.friends =
          [{name:'John', phone:'555-1212', age:10},
           {name:'Mary', phone:'555-9876', age:19},
           {name:'Mike', phone:'555-4321', age:21},
           {name:'Adam', phone:'555-5678', age:35},
           {name:'Julie', phone:'555-8765', age:29}];
      $scope.predicate = '-age';
    }]);
</script>
<div ng-app="orderByExample">
<div ng-controller="ExampleController">
  <pre>Sorting predicate = {{predicate}}; reverse = {{reverse}}</pre>
  <hr/>
  [ <a href="" ng-click="predicate=''">unsorted</a> ]
  <table class="friend">
    <tr>
      <th><a href="" ng-click="predicate = 'name'; reverse=false">Name</a>
          (<a href="" ng-click="predicate = '-name'; reverse=false">^</
a>)</th>
      <th><a href="" ng-click="predicate = 'phone'; reverse=!
reverse">Phone Number</a></th>
      <th><a href="" ng-click="predicate = 'age'; reverse=!reverse">Age</
a></th>
    </tr>
    <tr ng-repeat="friend in friends | orderBy:predicate:reverse">
      <td>{{friend.name}}</td>
      <td>{{friend.phone}}</td>
      <td>{{friend.age}}</td>
    </tr>
  </table>
</div>
</div>
```

La documentación de AngularJs dice que `expression` puede ser una función que podemos getionar, por ejemplo, en nuestro controlador. Ejemplo[37]:

```
  angular.module('orderByExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.friends =
```

[36] http://codepen.io/alexsuch/pen/XJgMzK
[37] http://codepen.io/alexsuch/pen/gbRmXd

```
          [{name:'John', phone:'555-1212', age:10},
           {name:'Mary', phone:'555-9876', age:19},
           {name:'Mike', phone:'555-4321', age:21},
           {name:'Adam', phone:'555-5678', age:35},
           {name:'Julie', phone:'555-8765', age:29}];

      var predicate = 'age';
      var reverse = false;

      $scope.predicate = predicate;
      $scope.reverse = reverse;

      $scope.setPredicate = function(_predicate){
        if(predicate === _predicate) {
          reverse = !reverse;
        } else {
          predicate = _predicate;
          reverse = false;
        }

        $scope.predicate = predicate;
        $scope.reverse = reverse;
      };

      $scope.getPredicate = function(){
        return (reverse?'-':'') + predicate;
      };
    }]);
```

```
<div ng-app="orderByExample">
<div ng-controller="ExampleController">
  <pre>Ordenando por = {{predicate}}; reverse = {{reverse}}</pre>
  <hr/>
  [ <a href="" ng-click="setPredicate(null)">unsorted</a> ]
  <table class="friend">
    <tr>
      <th><a href="" ng-click="setPredicate('name')">Name</a></th>
      <th><a href="" ng-click="setPredicate('phone')">Phone Number</a></
th>
      <th><a href="" ng-click="setPredicate('age')">Age</a></th>
    </tr>
    <tr ng-repeat="friend in friends | orderBy:getPredicate():reverse">
      <td>{{friend.name}}</td>
      <td>{{friend.phone}}</td>
      <td>{{friend.age}}</td>
    </tr>
  </table>
</div>
</div>
```

5.3. Cómo crear un filtro personalizado

Para crear un filtro, hay que registrar una nueva función factoría de tipo `filter` en nuestro módulo. Esta factoría debe devolver una función, que recibe como parámetro el elemento de entrada. Se pueden pasar los parámetros adicionales que haga falta.

En el siguiente ejemplo, construiremos un filtro que se encargue de formatear un número de teléfono [38]:

```
angular
 .module('telephoneSample', [])
 .filter('telephone', function(){
   return function(input) {
     var number = input || '';
     number = number.trim().replace(/[-\s\(\)]/g, '');

     if(number.length === 11) {
       var area = ['(', '+', number.substr(0,2) ,')'].join('');

       var local = [number.substr(2, 3), number.substr(5, 3),
number.substr(8, 11)].join('-');

       return [area,local].join(' ')
     }

     if(number.length === 9) {
       var local = [number.substr(0, 3), number.substr(3, 3),
number.substr(6, 11)].join('-');

       return local
     }

     return number;
   };
 });
```

Para probarlo en nuestra vista:

```
<div ng-app="telephoneSample">
  <div>{{'965123' | telephone}}</div>
  <div>{{'965123456' | telephone}}</div>
  <div>{{'34965123456' | telephone}}</div>
</div>
```

Para crear un filtro parametrizado, actuaríamos de la misma manera. El siguiente ejemplo muestra cómo sería un filtro que introduce un valor por defecto en caso de que el elemento que pasemos esté vacío. Éste texto será '-', o bien la cadena que pasemos como primer parámetro [39]:

```
angular
.module('textOrDefaultSample', [])
.controller('mainCtrl', function($scope){
  $scope.data = {
    nullValue : null,
    notNullValue: 'Hello world'
  };
```

[38] http://codepen.io/alexsuch/pen/prqtn
[39] http://codepen.io/alexsuch/pen/cnmlJ

```
})
.filter('textOrDefault', function () {
  return function (input, defaultValue) {
   defaultValue = defaultValue || '-';
   if (!input) {
     return defaultValue;
   }

   if (!angular.isString()) {
     if (input.toString) {
       input = input.toString();
     } else {
       return defaultValue;
     }
   }

   if (input.trim().length > 0) {
     return input;
   }

     return defaultValue;
   };
 });
```

```
<div ng-app="textOrDefaultSample" ng-controller="mainCtrl">
  <div>{{ data.nullValue | textOrDefault}}</div>
  <div>{{ data.nullValue | textOrDefault:'N/D'}}</div>
  <div>{{ data.notNullValue | textOrDefault}}</div>
  <div>{{ data.notNullValue | textOrDefault:'N/D'}}</div>
</div>
```

Vemos que hemos hecho uso de la función `angular.isString`. El objeto `angular` nos proporciona una serie de utilidades que pueden sernos de gran ayuda, y que podemos ver listados en https://docs.angularjs.org/api/ng/function.

5.4. *Tip*: cómo acceder a un array filtrado fuera de `ng-repeat`

Como hemos dicho varias veces, un filtro genera una salida nueva, sin modificar el elemento original. Así, supongamos el siguiente ejemplo [40]:

```
angular
  .module('app', [])
  .controller('mainCtrl', function($scope){
    $scope.teachers = [
      'Domingo',
      'Otto',
      'Eli',
      'Miguel Ángel',
      'Aitor',
      'Fran',
      'José Luís',
      'Álex'
```

[40] Demo en http://codepen.io/alexsuch/pen/qJCpd

```
    ];
  });
```

```
<div ng-app="app" ng-controller="mainCtrl">
<label>Filtrar: <input type="text" ng-model="search" /></label>
<ul>
  <li ng-repeat="it in teachers | filter:search">{{it}}</li>
</ul>
  Profesores filtrados: {{ teachers.length }}.
</div>
```

El resultado de *profesores filtrados*, filtremos los ítems que filtremos, siempre será 8.

Una cosa que se nos podría ocurrir para corregir esto, sería volver a filtrar el resultado:

```
Profesores filtrados: {{ teachers.length | filter:search }}.
```

Es una opción válida, pero no óptima: recordemos que un filtro es una operación que puede resultar bastante costosa, en función de lo que hayamos programado. Además, podemos encadenar filtros, lo que hace que su complejidad aumente en cada filtro:

```
Profesores filtrados: {{ teachers.length | filter:search | sort |
 uppercase | prepend:'--->' | append: '<---' }}.
```

Lo ideal para este caso sería emplear el mismo array filtrado para ambos casos. Y lo podemos conseguir, simplemente inicializando una variable al valor de los elementos filtrados [41] :

```
<div ng-app="app" ng-controller="mainCtrl">
<label>Filtrar: <input type="text" ng-model="search" /></label>
<ul>
  <li ng-repeat="it in filteredTeachers = (teachers |
 filter:search)">{{it}}</li>
</ul>
  Profesores filtrados: {{ filteredTeachers.length }}.
</div>
```

Para el caso de los filtros encadenados que hemos visto antes, sería exactamente igual:

```
<div ng-app="app" ng-controller="mainCtrl">
<label>Filtrar: <input type="text" ng-model="search" /></label>
<ul>
  <li ng-repeat="it in filteredTeachers = ( | filter:search | sort |
 uppercase | prepend:'--->' | append: '<---')">{{it}}</li>
</ul>
  Profesores filtrados: {{ filteredTeachers.length }}.
</div>
```

[41] Demo en http://codepen.io/alexsuch/pen/olJnD

5.5. Ejercicios

Aplica el *tag* `filters` a la versión que quieres que se corrija.

Adición de filtro de moneda (0,66 puntos)

Modifica el ejemplo del carrito de la compra para que los importes se formateen en euros y formato español de decimales. Revisa la documentación del filtro `currency`, porque tendremos que asegurarnos que siempre se muestren dos decimales, haya más o menos en el importe determinado para un prodcto en el servicio.

Custom filter (0,67 puntos)

Vamos a crear un filtro propio que nos devuelva el nombre del producto. Este filtro se llamará `productFullName`.

Recibirá como parámetro un objeto de tipo producto, con lo que en la vista lo pasaremos de la siguiente manera:

```
<p>
  {{ product | productFullName }}
  <br/>
  {{ product.price }} &euro;
</p>
```

Este filtro mostrará, concatenados, la marca y el nombre del producto. La marca deberá mostrarse en mayúsculas, y no deberemos usar la función `toUpperCase()` de JavaScript, sino inyectar el filtro `upperCaseFilter` en nuestro filtro y hacer uso de él.

Custom filter con parámetros (0,67 puntos)

Vamos a crear un filtro personalizado para la puntuación de los productos. Mostrará la puntuación en forma de estrellas. Se mostrarán tantas estrellas rellenas como puntuación tenga el producto, y luego se mostrarán estrellas vacías hasta llegar a un total de cinco. Por ejemplo, una puntuación de 3 mostrará ★★★☆☆

Para ello, vamos a crear un filtro de propósito general. Éste, por tanto, no recibirá un objeto como parámetro sino un valor. También, recibirá un parámetro adicional, que será el máximo de puntuación del producto. En este caso la puntuación máxima son cinco estrellas, pero de esta manera e filtro también nos valdrá para cuando puntuemos sobre diez.

Modificaremos nuestra plantilla para mostrar la puntuación debajo del importe:

```
<p>
  {{ product | productFullName }}
  <br/>
  {{ product.price }} &euro;
  <br/>
  {{ product.rating | rating:5 }}
</p>
```

Para el ejercicio, pegaremos directamente el carácter de las estrellas, en lugar de emplear su código HTML[42]. Si quisiéramos usar código html, tendríamos que hacer uso del servicio `$sce`[43] y de la directiva `ngBindHtml`[44]:

```
<p>
  {{ product | productFullName }}
  <br/>
  {{ product.price }} &euro;
  <br/>
  <span ng-bind-html="product.rating | rating:5"></span>
</p>
```

```
.filter('rating', function($sce){
    return function(rating,maxVal){
      var result = '';
      var solidStar = "&#9733;"; //#
      var outlineStar = "&#9734;" //#

      //TODO: implement filter

      return $sce.trustAsHtml(result);
    };
  })
```

[42] http://www.edlazorvfx.com/ysu/html/ascii.html
[43] https://docs.angularjs.org/api/ng/service/$sce
[44] https://docs.angularjs.org/api/ng/directive/ngBindHtml

6. Routing con `ngRoute`

Al principio de la sesión introductoria, vimos que todo el código estaba en la página principal y no había navegación hasta que se introdujo el módulo `ngRoute`, que nos permitió convertir nuestra aplicación en una SPA.

Desde la versión 1.2 de AngularJS, el módulo ngRoute no forma parte del *core* con lo que habrá que importarlo y añadirlo como dependencia de nuestra aplicación.

6.1. Gestión de la navegación

Para gestionar la navegación, AngularJS utiliza un truco, que consiste en que modifica partes de la barra de direcciones de una URL. ¿Cuáles? aquellas que van detrás del carácter `#`, también llamado *hashbang*. Todo lo que modifiquemos por detrás de este carácter se llama *URL fragment*. La especificación dice que, si cambiamos este fragmento sin alterar nada de lo que va por delante, el navegador no recargará la página. Sin embargo, este cambio sí que se guarda en el histórico del navegador, con lo cual los botones de atrás y adelante de nuestro navegador sí que funcionan. Será cosa nuestra gestionarlo correctamente para que la aplicación funcione como se desea al hacer uso de ellos.

Supongamos una serie de URLS de tipo CRUD. Idealmente, tendremos una URL para una lista de ítems, un formulario de edición, a lo mejor otro de creación, etc. Así, podrían ser:

- `/admin/users/list` – Para mostrar un listado de usuarios
- `/admin/users/new` – Formulario para crear un nuevo usuario
- `/admin/users/[userId]+ – Formulario para editar un usuario, cuyo ID es igual a +userId`.

Podríamos traducir estas URLs parciales a URLs con fragmentos para una SPA, usando el truco del hashbang que hemos comentado:

- http://www.miaplicaciondegestion.com/#/admin/users/list
- http://www.miaplicaciondegestion.com/#/admin/users/new
- userId[45]

6.2. La directiva ngView

La directiva `ngView+ es esencial para el uso de rutas, y complementa al servicio +$route`, incluyendo la plantilla de la ruta actual en nuestro layout principal, que debe estar situado en nuestro fichero `index.html`. Cada vez que la ruta cambia, todo el contenido de esta etiqueta cambiará, en función de lo que hayamos configurado en nuestro servicio de routing.

La directiva ng-view se puede usar a modo de elemento, o bien como atributo de un elemento `div`. Sin embargo, por cuestiones de compatibiliad con IE7 se recomienda su uso como atributo.

En el siguiente ejemplo se muestran las dos variantes, habiendo dejado comentado su uso como elemento.

[45] http://www.miaplicaciondegestion.com/#/admin/users/

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/
bootstrap/3.2.0/css/bootstrap.min.css"/>
</head>
<body ng-app="routing">

<div class="container">
    <div ng-include="'templates/common/header.html'"></div>
    <div ng-view></div>
    <!-- <ng-view></ng-view> -->
    <div ng-include="'templates/common/footer.html'"></div>
</div>

<script src="https://code.angularjs.org/1.2.22/angular.js"></script>
<script src="https://code.angularjs.org/1.2.22/angular-route.js"></script>
<script src="app.js"></script>
</body>
</html>
```

ngInclude

En el fragmento de código anterior, hemos visto dos bloques que hacen uso de la directiva `ngInclude`. Ésta se encarga de obtener un fragmento de HTML, compilarlo e introducirlo en nuestra aplicación.

Es muy útil para insertar elementos parciales que se van a repetir a lo largo de nuestra aplicación. Por ejemplo, en el bloque anterior se ha utilizado para añadir la cabecera y el pie de la aplicación. Son dos elementos que van a estar siempre ahí, y no queremos "ensuciar" nuestro fichero `index.html` con su código.

Aquí[46] tenemos un enlace de una aplicación en ese estado.

Fijáos que la directiva recibe como parámetro una expresión, por eso su valor está entre comillas simples.

6.3. Definición de rutas

En AngularJS, las rutas se definen en la fase de configuración de la aplicación, haciendo uso del servicio `$routeProvider`. Éste proporciona una API sencilla donde podemos encadenar métodos para definir rutas (método `when`), y establecer una ruta por defecto (`otherwise`).

El método `when` recibe como entrada dos parámetros:

- `path` (string). Ej: /user/list
- `ruta` (objeto). Puede tener varios atributos [47], pero lo normal es que usemos:

[46] https://bitbucket.org/alejandro_such/angularjs-routing-examples/src/7789079fea7cb3677c507f7e7067ecfa0c74ec5e?at=v1.0

[47] Podemos ver toda la configuración en https://docs.angularjs.org/api/ngRoute/provider/$routeProvider

`controller`. Será el nombre de un controlador que hayamos creado en nuestra aplicación. Al definir aquí el controlador, ya nos ahorramos el tener que emplear la directiva `ng-controller en nuestro código.
`templateUrl`. Ruta hacia la plantilla HTML con el contenido de nuestro parcial.

Por su parte, el método `otherwise` suele recibir un parámetro, consistente en un objeto con un atributo `redirectTo`, indicando la URL a la que redirigir cuando no se encuentra ninguna ruta concordante.

Supongamos una aplicación de dos páginas. Consiste en una aplicación de pedidos, donde en la primera página tenemos un listado de pedidos, y en la segunda un formulario para realizar nuevos pedidos. La página por defecto de nuestra aplicación será el listado de pedidos.

- Un índice, que contiene un listado de todos los pedidos realizados.
- Un formulario de introducción de nuevos pedidos.

Para no incrementar la complejidad de nuestra aplicación, los pedidos consistirán en una cadena de texto. Aquí[48] tenemos el código completo de esta aplicación. Veamos cómo hemos configurado las rutas:

```
angular
    .module('ordersapp', ['ngRoute'])
    .config(function($routeProvider){
        $routeProvider
            .when('/orders', {
                templateUrl:'orders/tpl/list.html',
                controller: 'OrdersCtrl'
            })
            .when('/orders/new', {
                templateUrl: 'new-order/tpl/new.html',
                controller: 'NewOrderCtrl'
            })
            .otherwise({ redirectTo : '/orders'});
    });
```

Si nos vamos a una de las vistas, por ejemplo `orders/tpl/list.html` vemos que no se ha introducido la directiva `ng-controller` al haber definido el controlador en la definición de rutas.

```
<h2>Order list</h2>
<div class="row" ng-repeat="order in orders">
    <div class="col col-xs-12">
        <p>{{ order }}</p>
    </div>
</div>

<div class="row">
    <div class="col col-xs-12">
        <a href="#/orders/new" class="btn btn-default">New order</a>
    </div>
```

[48] https://bitbucket.org/alejandro_such/angularjs-routing-examples/src/22fe7b8441e74be3e2352e1c0034f0376f6014b6/?at=v2.0

```
</div>
```

6.4. Rutas parametrizadas

En la aplicación que hemos hecho, utilizamos un sistema de rutas bastante sencillo, que no utiliza ninguna parte variable en la URL. Sin embargo, hoy en día estamos hartos de ver URLs con partes variables. Los antiguos *search parameters*

```
/users/edit/id?=1
/users/edit/id?=2
/users/edit/id?=114
```

Ha pasado de moda, y ahora lo habitual es construir URLs de la forma

```
/users/edit/1
/users/edit/2
/users/edit/114
```

Hacer esto con el sistema de *routing* de AngularJS es bastante sencillo: podemos declarar elementos variables simplemente poniendo el símbolo de los dos puntos (`:`) delante de éste. Así, en nuestra aplicación de pedidos, introduciremos una ruta nueva para ver el detalle de un pedido.

```
angular
    .module('ordersapp', ['ngRoute'])
    .config(function($routeProvider){
        $routeProvider
            .when('/orders', {
                templateUrl:'orders/tpl/list.html',
                controller: 'OrdersCtrl'
            })
            .when('/orders/new', {
                templateUrl: 'new-order/tpl/new.html',
                controller: 'NewOrderCtrl'
            })
            .when('/orders/edit/:idx', { //Ruta parametrizada
                templateUrl: 'view-order/tpl/view.html',
                controller: 'ViewOrderCtrl'
            })
            .otherwise({ redirectTo : '/orders'});
    });
```

Para recibir los parámetros en nuestro controlador, deberemos hacer uso del servicio `$routeParams`. Éste nos permite recibir todos los parámetros que hayamos pasado a la URL.

```
angular
    .module('ordersapp')
    .controller('ViewOrderCtrl', function ($scope, OrdersService,
 $routeParams){
        $scope.order = OrdersService.getOrder($routeParams.idx);
```

```
});
```

Lo bueno del servicio `$routeParams` es que combina los parámetros que llegan tanto por URL como por *search parameters*. Con lo cual, hubiera funcionado exactamente igual, y sin tocar nada, de haber utilizado una url de la forma `orders/edit?idx=3`

Aquí[49] tenemos el código completo de la aplicación de pedidos.

6.5. Redirección

En el código de ejemplo ya hay bloques que muestran cómo realizar una redirección de una página a otra. No obstante lo mencionaremos más detenidamente, con un par de recomendaciones.

Desde una vista

Para navegar de una ruta a otra desde una vista, lo haremos de manera exactamente igual a como lo haríamos en HTML: con la etiqueta `<a>`. No obstante, los destinos irán todos precedidos por el *hashbang*. Por ejemplo:

```
<a href="#/users/list">Listado de usuarios</a>
```

Para referenciar una ruta variable, podemos hacerlo de igual manera, haciendo uso de las variables de AngularJS:

```
<a href="#/users/edit/{{user.id}}">Editar usuario</a>
```

Sin embargo, cuando utilizamos código dinámico (variables, funciones, etc.) en un enlace, la manera más correcta de hacerlo es utilizando la directiva `ngHref`.

En algunas ocasiones al generar el enlace dinámico anterior, podría darse el caso de que el usuario hace clic antes de que AngularJS haya tenido la oportunidad de establecer el valor dinámico. Al usar la directiva `ngHref`, AngularJS no establece el valor del atributo `href` hasta que ha podido interpretar el valor completo de la cadena. Así, lo que hace es traducir esto:

```
<a ng-href="#/users/edit/{{user.id}}">Editar usuario</a>
```

a esto otro:

```
<a href="#/users/edit/1">Editar usuario</a>
```

en cuanto tiene la mínima oportunidad.

Desde un controlador. El objeto $location

En no pocas ocasiones querremos que la redirección se haga en función de que cierta lógica de negocio se haya aplicado correctamente o no. Esto implica que dicha redirección tenga

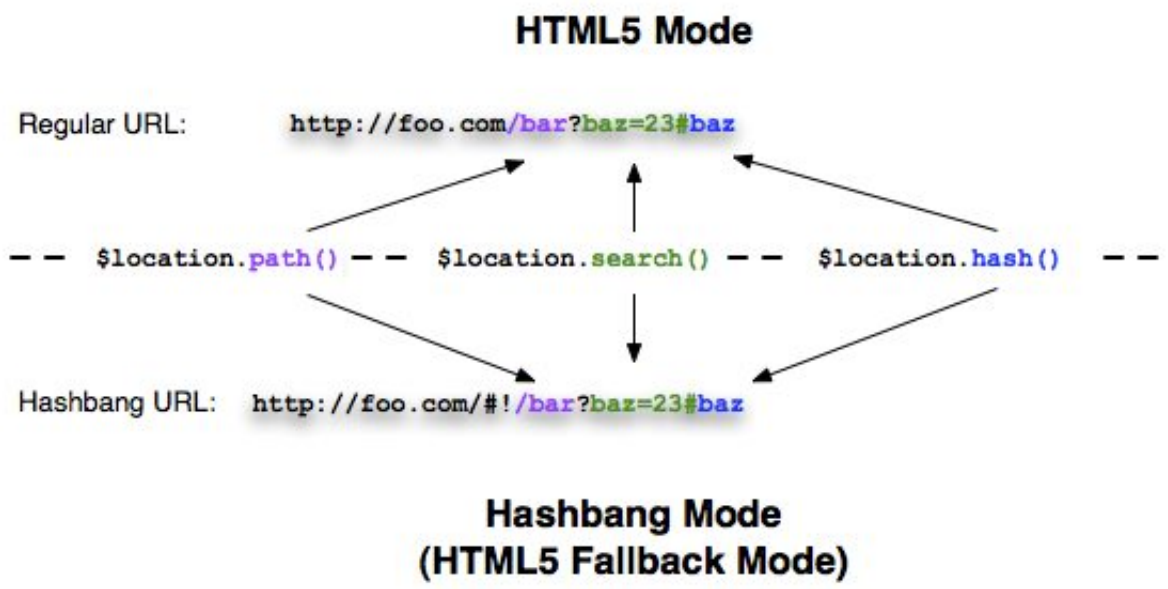
[49] https://bitbucket.org/alejandro_such/angularjs-routing-examples/src/4124f251b15d83dd793bd0d85df7979faa16bb8a/?at=v3.0

que hacerse desde un controlador u otro servicio. Para hacerlo desde aquí, recurriremos al servicio `$location`.

El servicio `$location`+ [50] `parsea la URL de la barra de direcciones del navegador (según los valores de +window.location)`, y hace que la URL esté accesible para nuestra aplicación. Todo cambio que se haga en la URL se verá reflejado en el servicio `$location` y viceversa.

El servicio `$location`:

- Expone la URL actual de la barra de direcciones del navegador, para que podamos:
  # Observar la URL
  # Modificar la URL
- Sincroniza la URL de la barra de direcciones del navegador cuando el usuario:
  # Modifica la barra de direcciones
  # Hace clic en los botones *forward* o *back* del navegador (o hace click en un enlace histórico)
  # Hace clic en un enlace.
- Representa el objeto URL como un conjunto de métodos:
  # `protocol`: http, https,…
  # `host`: localhost, dccia.ua.es
  # `port`: 80, 8080, 63342,…
  # `path`: /orders
  # `search`
  # `hash`



[50] https://docs.angularjs.org/api/ng/service/$location

Estos métodos actúan a la vez como getters y setters, en función de si tienen o no parámetros. Así hemos podido ver cómo en el controlador `NewOrderCtrl`, redirigíamos a la página principal una vez insertado un pedido mediante la orden:

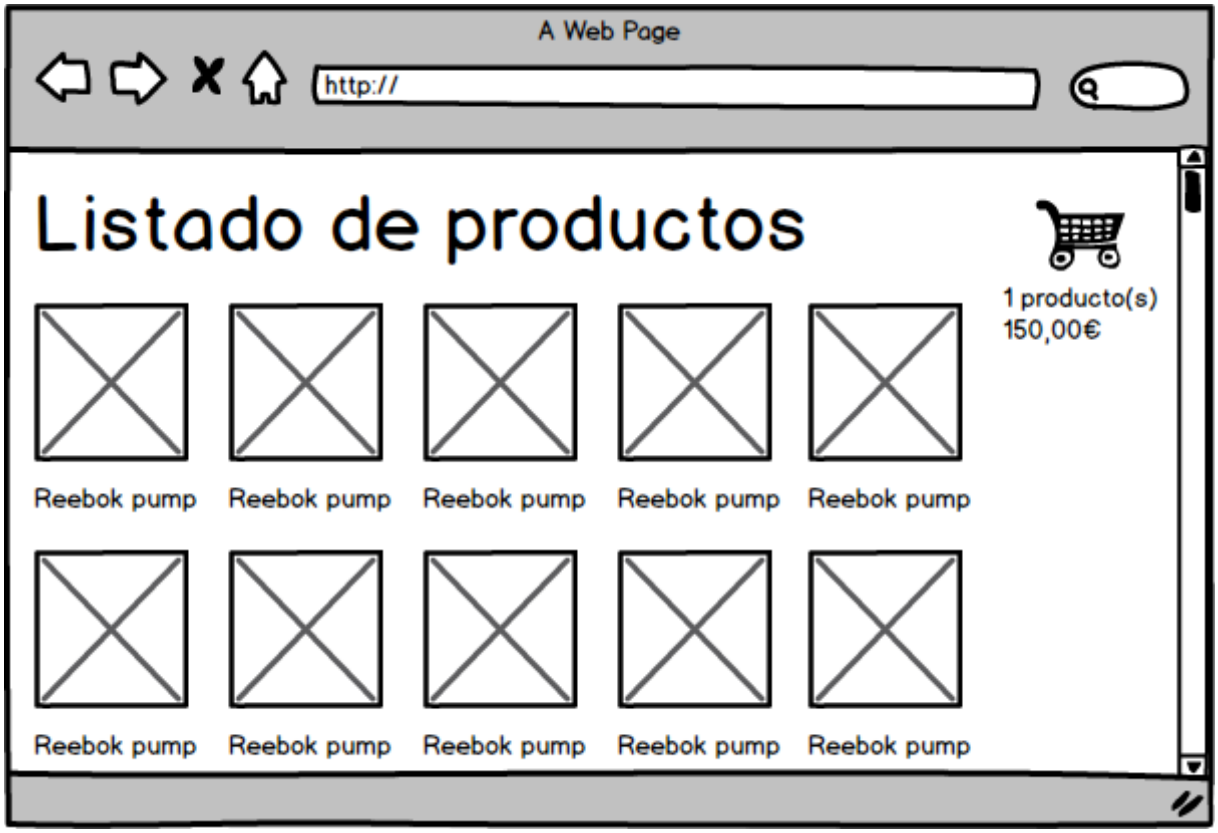
```
$location.path('/orders');
```

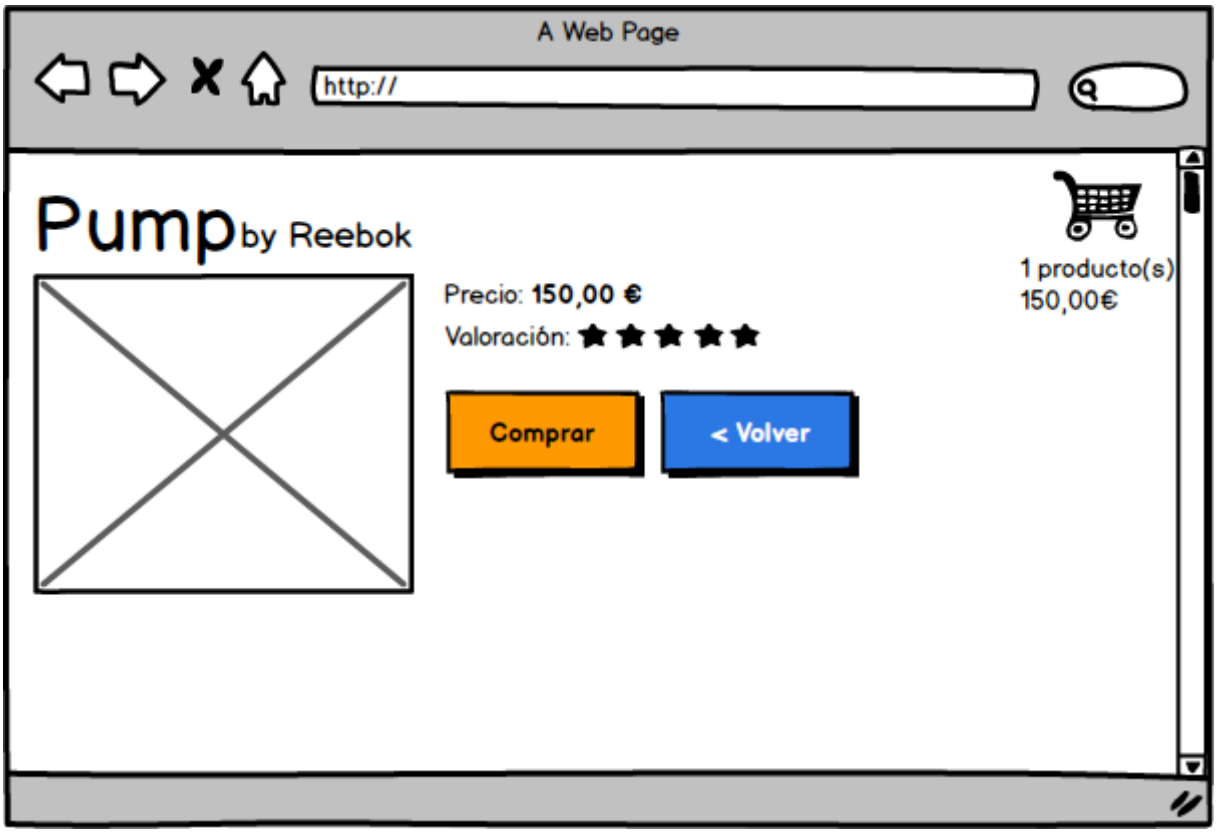
6.6. Ejercicio (1 punto)

Modifica el ejemplo de los pedidos para adaptar el carrito de la compra que hemos estado haciendo en las últimas sesiones.

Tendremos de igual manera dos pantallas: una de listado (ruta: `/list`) y otra de detalle de producto (ruta: `/detail/:id_producto`). La ruta por defecto será la del listado.

Como vemos en los siguientes *mockups*, la cabecera tendrá un título genérico, o bien el nombre del producto. Además, tendrá la cesta de la compra indicando el número de productos y el importe total.





Aplica el *tag* `ngRoute` a la versión que quieres que se corrija. <<<

7. Routing con `ui-router`

Una cosa que puede no parecer muy obvia, es que el routing de URLs puede considerarse como una máquina de estados finitos. Cuando configuramos las rutas, estamos definiendo los distintos estados por los que atraviesa nuestra aplicación, e informando a la aplicación qué debe mostrarse cuando estamos en una ruta determinada.

Hemos visto que AngularJS nos proporciona un mecanismo de routing que, pese a ser totalmente válido, tiene ciertas limitaciones. Entre ellas, en la clase anterior hemos visto la necesidad de incluir, en cada una de las vistas, la cabecera y el pie con directivas `ngInclude`. Además, las redirecciones se hacían directamente contra la ruta de manera que, si esta cambia, debemos ir a cada etiqueta `a` y cada llamada a `$location.path()` a modificarla.

El módulo `ui-rouoter` se adapta perfectamente al concepto de routing como máquina de estados finita. Permite definir estados, y transiciones de un estado a otro. Además, nos permite desacoplar estados anidados, y gestionar layouts más complejos de una manera sencilla y elegante.

El concepto de routing es un poco distinto, pero a la larga acaba gustando más que el de `ngRoute`

En este capítulo, vamos a modificar la aplicación de pedidos realizada en el capítulo anterior y adaptarla a `ui-router`.

En nuestra aplicación, identificamos un layout con tres componentes:

- Cabecera
- Cuerpo

- Pie

7.1. Primeros cambios en la aplicación

Lo primero que haremos será deshacernos del módulo `ngRoute`, e incluir el módulo de `ui-router`, para ello, eliminaremos la línea

```
<script src="https://code.angularjs.org/1.2.22/angular-route.js"></script>
```

Y en su lugar introduciremos la siguiente:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.2.10/
angular-ui-router.js"></script>
```

Deberemos inyectar, además, el módulo `ui.router` en nuestra aplicación:

```
angular
    .module('ordersapp', ['ui.router'])
```

7.2. La directiva `uiView`

Al igual que con `ngRoute` era imprescindible el uso de la directiva `ngView` para declarar dónde iba el contenido de cada ruta en la vista, aquí haremos uso de la directiva `uiView`.

Sin embargo, una de las ventajas de `ui-router` es que nos permite definir más de un bloque de este tipo, por lo que vamos a definir tres de ellos:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/
bootstrap/3.2.0/css/bootstrap.min.css"/>
</head>
<body ng-app="ordersapp">

<div class="container">
    <div ui-view name="header"></div>
    <div ui-view name="content"></div>
    <div ui-view name="footer"></div>
</div>

<script src="https://code.angularjs.org/1.2.22/angular.js"></script>
<script src="//cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.2.10/
angular-ui-router.js"></script>
<script src="app.js"></script>
<script src="components/services/OrdersService.js"></script>
<script src="orders/controllers/OrdersCtrl.js"></script>
<script src="new-order/controllers/NewOrderCtrl.js"></script>
<script src="view-order/controllers/ViewOrderCtrl.js"></script>
</body>
```

```
</html>
```

Al realizar este cambio, vamos a olvidarnos de tener que realizar un `ng-include` en cada una de las vistas.

Al igual que teníamos una plantilla llamada `header` y otra llamada `footer`, crearemos ahora una tercera plantilla, `content.html`, cuyo contenido será:

```
<div class="row">
    <div class="col col-xs-12" ui-view>
        <h4>Welcome to the orders app</h4>
    </div>
</div>
```

Vemos que éste también tiene una directiva `ui-view`. En seguida veremos por qué.

7.3. Definiendo nuestro primer estado

Nuestro primer estado se corresponderá con la ruta /orders.

Al igual que en el capítulo anterior hacíamos uso del servicio `$routeProvider` para la definición de rutas, aquí utilizaremos el servicio `$stateProvider`, ya que hemos dicho que consideraremos nuestro sistema de *routing* como una máquina de estados.

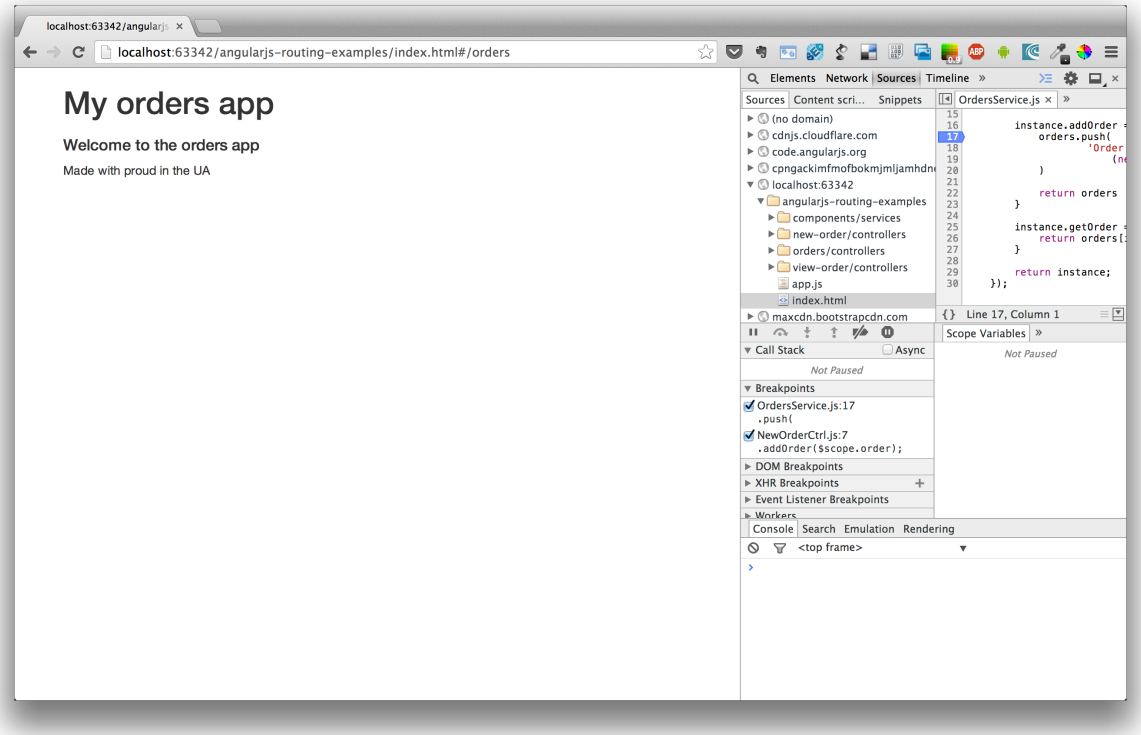
Para ello, el servicio $stateProvider dispone de un método llamado `state`, que recibe como primer parámetro un nombre de estado (el que nosotros queramos), y como segundo parámetro un objeto con los atributos:

- `url`: url del estado que estamos definiendo
- `views`: objeto que tendrá tantos atributos como directivas `ui-view` hayamos definido. En nuestro caso habrá tres (`header`, `content`, `footer`). Al igual que en el caso de `ngRoute`, aquí podremos definir el `templateUrl` para la vista a cargar, y un `controller` para definir el controlador que gestionará dicha vista. Como de momento no vamos a querer un controlador, no lo definimos para ninguna de ellas.

```
angular
    .module('ordersapp', ['ui.router'])
    .config(function ($stateProvider) {
        $stateProvider
            .state('orders', {
                url: '/orders',
                views: {
                    header: {
                        templateUrl: 'components/templates/common/
header.html'
                    },
                    content: {
                        templateUrl: 'components/templates/common/
content.html'
                    },
                    footer: {
                        templateUrl: 'components/templates/common/
footer.html'
                    }
```

```
                }
            });
    });
```

Si vamos a http://localhost:63342/angularjs-routing-examples/index.html#/orders, veremos que el layout de nuestra aplicación ya está conformado.



7.4. Estados anidados

Hasta ahora, hemos creado el esqueleto de nuestra aplicación. No vemos dónde está el listado de pedidos. Para ello, definiremos un nuevos estado, llamado `orders.list`

```
.state('orders.list', {
    url: '/list',
    controller: 'OrdersCtrl',
    templateUrl: 'orders/tpl/list.html'
})
```

Si nos vamos ahora a la URL http://localhost:63342/angularjs-routing-examples/index.html#/orders/list, veremos que ya tenemos el listado de productos de igual manera que teníamos en el capítulo anterior.

A priori, nos llamará la atención varias cosas. La primera de ellas, es que en nuestro estado hemos definido la url como `list`, y en nuestra aplicación aparece `/orders/list` como URL.

Esto se debe a que, por definición, todo estado que tenga un nombre dado, precedido por el nombre de otro estado y un punto (`orders.list`) se considera un estado anidado (*nested state*).

Un estado anidado hereda todo lo definido en el estado padre. Su URL, además, será composición de la URL del padre, más la URL que en el estado definamos. Es por ello que la URl es `/orders/list`. Una gran ventaja que aporta es que, si queremos renombrar la URL a `order` (por poner un ejemplo), únicamente debemos hacerlo en un punto. Además, todo el layout del padre se hereda, por eso no hemos tenido que definir la cabecera ni el pie.

¿Pero cómo sabe `ui-router` dónde colocar la vista? Muy sencillo. Si volvemos a ver el código de la plantilla `content.html` veremos que ahí se había definido un objeto `div` con un atributo `ui-view`. Éste es el punto que aprovecha `ui-router` para introducir la nueva plantilla, dejando el resto intacto.

Además, como a este nivel ya disponemos sólo de un `ui-view`, no es necesario jugar con el objeto `views` como habíammos hecho en la definición del estado anterior: podemos definir el `controller` (aquí sí que necesitamos ya uno) y el `templateUrl` a nivel de raíz del objeto.

De igual manera, definiremos los estados de creación y detalle:

```
.state('orders.new', {
    url: '/new',
    templateUrl: 'new-order/tpl/new.html',
    controller: 'NewOrderCtrl'
})
.state('orders.edit', {
    url: '/edit/:idx',
    templateUrl: 'view-order/tpl/view.html',
    controller: 'ViewOrderCtrl'
})
```

7.5. Definiendo una ruta por defecto

Al igual que con el servicio `ngRoute` podíamos definir un estado por defecto en caso de no encontrar ninguna ruta, aquí también lo podemos hacer. Para ello, necesitamos inyectar el servicio `$urlRouterProvider` en nuestra función de configuración. Este servicio dispone de un método `otherwise`, que recibe como parámetro la URL destino a la que redirigir en caso de no haber resuelto ninguna.

```
angular
    .module('ordersapp', ['ui.router'])
    .config(function ($stateProvider, $urlRouterProvider) {

        // DEFINICIÓN DE ESTADOS

        $urlRouterProvider.otherwise('/orders/list');
    });
```

7.6. Estados abstractos

Puede que os hayáis preguntado si realmente es necesario tener una ruta `/orders` que sea accesible desde el navegador. Efectivamente, esta ruta nos ha valido para conformar el *layout* inicial y no la necesitamos para nada más, ya que no aporta nada en absoluto. El módulo `ui-router` contempla este caso, y nos permite definir el estado `orders` como un estado abstracto. Al igual que una clase java, un estado abstracto no puede generarse por sí sólo,

sino a través de alguna de las clases que lo extienden. Podemos declarar un estado como abstracto añadiéndole el atributo `abstract:true`:

```
.state('orders', {
    abstract: true,
    url: '/orders',
    views: {
        header: {
            templateUrl: 'components/templates/common/header.html'
        },
        content: {
            templateUrl: 'components/templates/common/content.html'
        },
        footer: {
            templateUrl: 'components/templates/common/footer.html'
        }
    }
})
```

Si intentamos ir ahora a http://localhost:63342/angularjs-routing-examples/index.html#/orders, veremos que la URL no se resuelve correctamente, con lo que seremos redirigidos al estado definido en `$urlRouterProvider.otherwise`.

7.7. Recepción de parámetros en el controlador

Para la recepción de parámetros en un controlador utilizaremos el servicio `$stateParams`, que funciona de igual manera que el servicio `routeParams`. Así, nuestro cambios en el controlador `ViewOrderCtrl` serán m##nimos.

```
angular
    .module('ordersapp')
    .controller('ViewOrderCtrl', function ($scope, OrdersService,
 $stateParams){
        $scope.order = OrdersService.getOrder($stateParams.idx);
    });
```

7.8. Redirección

A nivel de redirección, tendremos que hacer unos cambios mayores. En `ui-router`, en lugar de la ruta, indicaremos al estado al que queremos realizar la transición. Es muy fácil querer cambiar el nombre de una ruta. Sin embargo, los estados tienen una nomenclatura con un significado semántico, que no querremos cambiar. Ahora, si decidimos traducir nuestras URLs a español, sólo tendremos que hacerlo a nivel de configuración.

Desde una vista

Desde una vista, cambiaremos nuestros `ng-href="route"` por `ui-sref="state"`.

En caso de incluir parámetros, añadiremos un objeto con el nombre de el(los) parámetro(s).

Por ejemplo, la plantilla `orders/tpl/list.html` quedará:

```
<div>
```

```
    <h2>Order list</h2>
    <div class="row" ng-repeat="order in orders">
        <div class="col col-xs-12">
            <p>{{ order }} <a ui-sref="orders.edit({idx:$index})">[Edit]</
a></p>
        </div>
    </div>

    <div class="row">
        <div class="col col-xs-12">
            <a ui-sref="orders.new" class="btn btn-default">New order</a>
        </div>
    </div>
</div>
```

Desde un controlador

Desde un controlador, haremos uso del servicio `state`, que dispone del método `go(stateName)`. La variación a realizar sobre el controlador `NewOrderCtrl` sería:

```
angular
    .module('ordersapp')
    .controller('NewOrderCtrl', function ($scope, OrdersService, $state) {
        $scope.order = null;

        $scope.saveOrder = function(){
            OrdersService.addOrder($scope.order);
            $state.go('orders.list');
        };
    });
```

En caso de querer redirigir a una ruta con parámetros, los pasaremos en un objeto JSON:

```
$state.go('orders.list', {idx: 0});
```

Aquí[51] tenéis acceso al código de la aplicación de pedidos modificada y adaptada a `ui-router`.

7.9. Ejercicio (1 punto)

Adapta el ejemplo de la sesión anterior para utilizar ui-router.

Aplica el *tag* `uiRouter` a la versión que quieres que se corrija. <<<

8. Formularios y validación

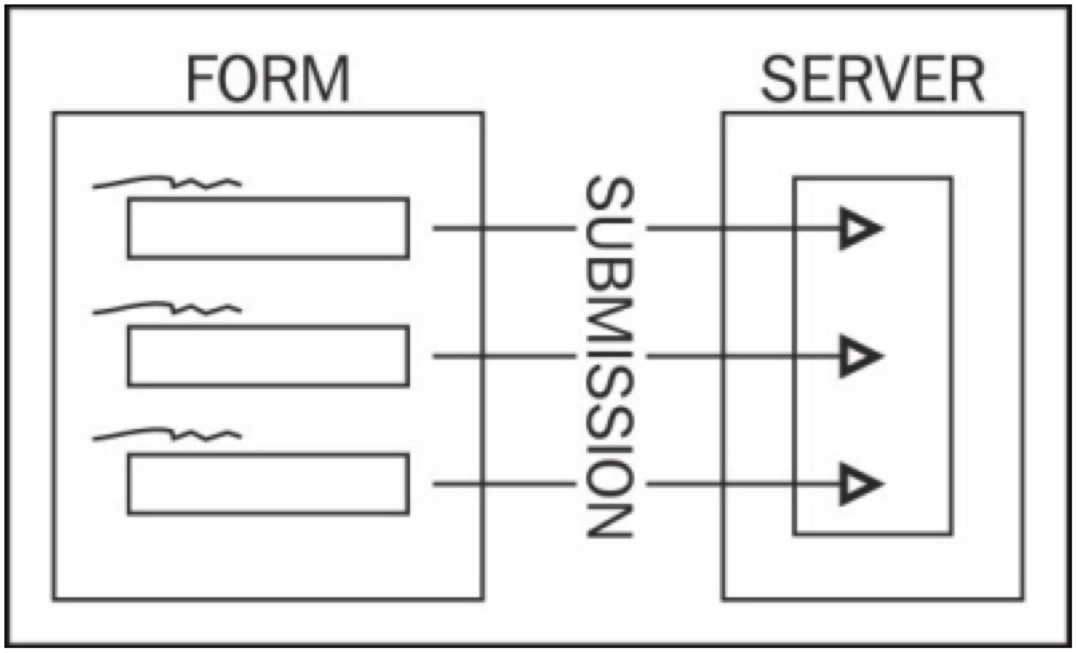
AngularJS se basa en formularios HTML e inputs estándar. Esto quiere decir que podemos seguir creando nuestra UI a partir de los mismos elementos que ya conocemos, usando herramientas de desarrollo HTML estándar.

[51] https://bitbucket.org/alejandro_such/angularjs-routing-examples/src/44799f9217ea37318564eae04194481d70eb43e0/?at=v4.0

8.1. Comparando formullarios tradicionales con formularios en AngularJS

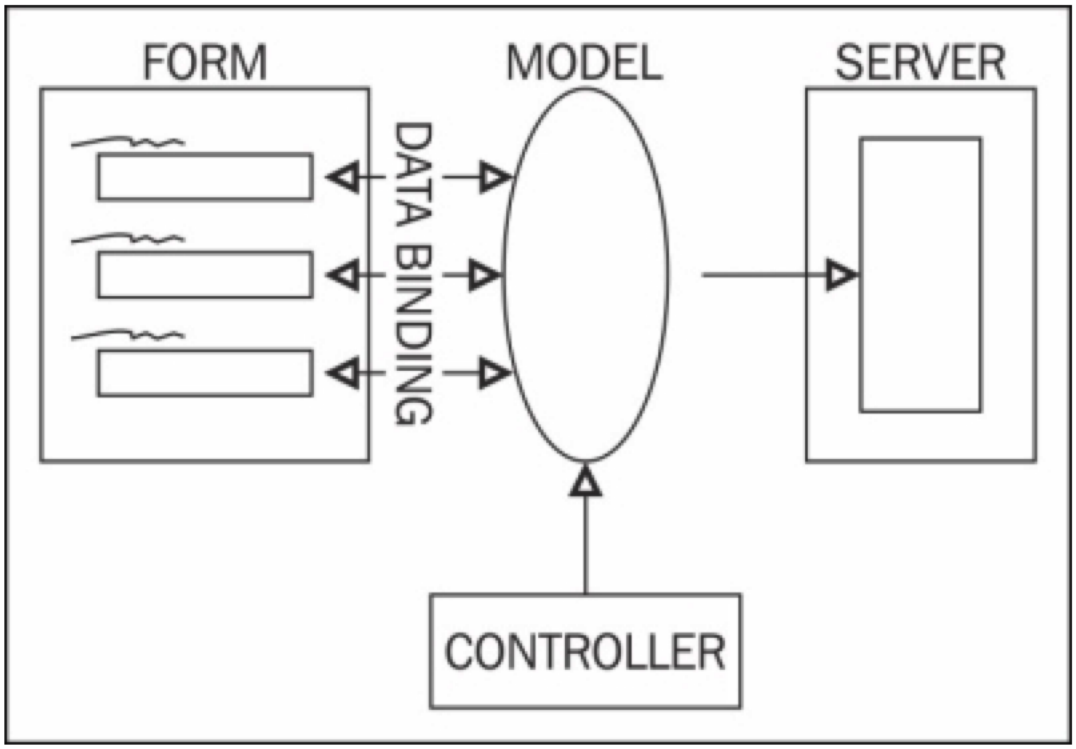
Vamos a ver cómo funcionan los formularios en AngularJS, y cómo éste modifica y extiende el comportamiento de los inputs de HTML, y cómo gestiona las actualizaciones del modelo. También veremos las directivas de validación incluidas en el *core* de AngularJS, para finalmente crear nuestras propias directivas de validación.

En un formulario HTML estándar, el valor de un input es el valor que se enviará al servidor al ejecutarse la acción *submit* del formulario.



El problema es que a veces, no queremos trabajar con los datos tal y como se muestran en el formulario. Por ejemplo, podríamos querer mostrar una fecha formateada (ej: 14 de julio de 2.012), pero lo más seguro es que queramos trabajar con un objeto JavaScript de tipo Date. Tener que realizar estas transformaciones constantemente es algo muy tedioso, y puede conducir a errores.

Al desacoplar el modelo de la vista en AngularJS, no nos tenemos que preocupar del valor del modelo cuando éste cambia en la vista, ni del tipo de dato cuando trabajamos con él en un controlador.



Esto se consigue a través de las directivas `form` e `input`, así como con las directivas de validación y los controladores. Estas directivas de validación sobreescriben el comportamiento por defecto de los formularios HTML. Sin embargo, mirando su código, vemos que son prácticamente iguales que los formularios HTML estándar.

En primer lugar, la directiva `ngModel` nos permite definir cómo los `input` se deben asociar (*bind*) al modelo.

Hemos visto cómo AngularJS crea un databinding entre los campos del objeto `scope` y los elementos HTML en la página, usando dobles llaves `{{}}` y la directiva `ngBind`, que explicaremos ahora haciendo un inciso.

La directiva `ngBind`

En algunos navegadores, podemos experimentar cierto "parpadeo" de valores en AngularJS. Esto se debe a que primero se carga el HTML y luego el código AngualrJS. De este modo, es posible que veamos las variables entre llaves antes que sus valores.

A este fenómeno se le conoce como PRF (*Pre-Render Flickering*). Para evitarlo, se introdujo la directiva `ngBind`.

Para hacer uso de ella sólo tenemos que añadir el atributo `ng-bind` a un elemento, y escribir una expresión dentro de éste. Por ejemplo, en lugar de:

```
<h1>{{model.header.title}}</h1>
```

podemos usar:

```
<h1 ng-bind="model.header.title"></h1>
```

Si en nuestro html no teníamos ningún elemento para nuestro texto, siempre podemos utilizar un `<span>` para introducir ahí nuestra expresión. Como véis, es igual de sencillo que usar los corchetes dobles, y ayuda a prevenir el PRF.

8.2. Continuemos

Como decíamos, ya sabemos cómo se realiza el databinding con la doble llave o la directiva `ngBind`. Estas técnicas sólo permiten el *binding* en una dirección (*one-way binding*). Para asociar el valor de una directiva `input`, y así conseguir un *two-way data binding* usamos, además, la directiva `ngModel`. Veamos el siguiente ejemplo [52]:

```
<div ng-app="databinding" ng-controller="MainCtrl">
  <div>
    Hola, {{name}}!
  </div>
  <div>
    Hola, <span ng-bind="name"></span>!
  </div>
  <div>
    <label>Nombre: <input type="text" ng-model="name" /></label>
  </div>
</div>
```

```
angular
  .module('databinding', [])
  .controller('MainCtrl', function($scope){
    $scope.name = 'Alejandro';
  })
```

En los dos primeros `div`, *bindamos* el atributo `name` del `scope` con dobles llaves, mientras que en el segundo lo hacemos a través de la directiva `ng-bind`. Este *binding* se realiza únicamente en una dirección: si cambiamos el valor de `scope.name` en el controlador, éste cambiará en la vista. Sin embargo, no hay manera de cambiarlo en la vista y que esto afecte al controlador.

Sin embargo, en el último div, AngularJS *binda* el valor de `scope.name` al del elemento `input`, a través de la directiva `ngModel`. Aquí es donde se realiza un *two-way data binding*. Se puede observar sencillamente ya que, si modificamos el valor del input, los otros dos elementos modifican el texto.

Además, veremos que AngularJS permite que las directivas transformen y validen los valores de `ngModel` en el momento que se realiza el paso de valores de la vista al controlador.

8.3. Creando un formulario de registro

Para tratar estos temas, vamos a crear un formulario de registro, que tendrá los siguientes campos y restricciones.

- Nombre. Requerido. Longitud mínima de 3 caracteres y máxima de 25.
- Apellidos. Requerido. Mínimo dos palabras
- Email. Requerido. Email válido

[52] http://codepen.io/alexsuch/pen/gezjy

- Sexo. Requerido. Será un selector de tipo `radio`.
- Website. Requerido. Deberá ser una URL válida.
- Provincia. Requerido. Será un selector de tipo `select`.
- Suscripción a newsletter. Opcional. Tipo `checkbox`.

Nuestra primera aproximación sería la siguiente [53]:

```
<div ng-app="formExample" ng-controller="mainCtrl">

  <form ng-submit="">
    <div>
      <label>
        nombre: <input type="text" ng-model="user.name" />
      </label>
    </div>
    <div>
      <label>
        apellidos: <input type="text" ng-model="user.lastName" />
      </label>
    </div>
    <div>
      <label>
        email: <input type="text" ng-model="user.email" />
      </label>
    </div>
    <div>
      sexo:
      <label>
        hombre <input type="radio" value="h" ng-model="user.sex" />
      </label>
      <label>
        mujer <input type="radio" value="m" ng-model="user.sex" />
      </label>
    </div>
    <div>
      <label>website: <input type="text" ng-model="user.website" /></
label>
    </div>
    <div>
      provincia:
      <select ng-model="user.province">
        <option value="12">Castellón</option>
        <option value="46">Valencia</option>
        <option value="03">Alicante</option>
      </select>
    </div>
    <div>
      <label>
        suscribirse a la newsletter <input type="checkbox" ng-
model="user.newsletter" />
      </label>
    </div>
  </form>
  <pre ng-bind="user | json"></pre>
```

[53] http://codepen.io/alexsuch/pen/DkBmn

```
</div>
```

Campos requeridos

Usaremos la directiva `ngRequired` (o simplemente `required`) para especificar aquellos campos obligatorios. Así, todos aquellos campos cuyo valor sea `null`, `undefined` o una cadena vacía serán inválidos. Por ejemplo, el campo nombre es uno de los campos obligatorios:

```
<input type="text" ng-model="user.name" required />
```

Tamaño mínimo y máximo

En el campo nombre, también habíamos definido un tamaño mínimo y máximo. Esto lo conseguimos gracias a las directivas `ngMinlength` y `ngMaxLength`:

```
<input type="text" ng-model="user.name" required ng-minlength="3" ng-
maxlength="25" />
```

Expresiones regulares

Por su parte, el campo *apellidos*, además de ser obligatorio, tenía la restricción de que debía contar, al menos con dos palabras. Podemos definir esta restricción de manera sencilla con expresiones regulares. La directiva `ngPattern` se encarga de validar que un elemento cumpla con una expresión regular determinada:

```
<input type="text" ng-model="user.lastName" required ng-pattern="/^\w+(\s
\w`)`$/" />
```

Email

La validación de emails es muy sencilla. Simplemente debemos cambiar el `input type="text"` por `input  type="email"`. AngularJS ya se encargará realizar las validaciones necesarias para este tipo:

```
<input type="email" ng-model="user.email" required />
```

Radio buttons

Los `radiobuttons` proporcionan un grupo fijo de opciones para un campo. Son muy sencillos de implemntar. Sólo hay que asociar los `radiobutton` de un mismo grupo al mismo modelo. Se usará el atributo estándar `value` para determinar qué valor pasar al modelo. Así, el valor de sexo será:

```
<div>
  sexo:
  <label>
    hombre <input type="radio" value="h" ng-model="user.sex" required />
  </label>
```

```
  <label>
    mujer <input type="radio" value="m" ng-model="user.sex" required />
  </label>
</div>
```

URLs

Al igual que el `type="email"`, también disponemos de un `type="url"` para que la validación de URLs sea lo más sencilla posible:

```
<input type="url" ng-model="user.website" required /></label>
```

Selectores

La directiva `select` nos permite crear una *drop-down list* desde la que el usuario puede seleccionar uno o varios ítems. AngularJS nos permite especificar estas opciones de manera estática, como en un `select` HTML estándar, o de manera dinámica a partir de un array.

De hecho, es muy normal el uso de un array de objetos. Aquí para simplificar, hemos utilizado las tres provincias de la Comunidad Valenciana. Sería normal introducir las 50 provincias de España (más las dos ciudades autónomas de Ceuta y Melilla) a partir de los datos proporcionados por un servicio. Para simplificarlo, vamos a suponer que ya las tenemos en nuestro controlador:

```
$scope.provinces =  [
  { name : 'Castellón', code : '12'},
  { name : 'Valencia', code : '46'},
  { name : 'Alicante', code : '03'}
];
```

Para *bindar* el valor de este array a un elemento `select` tenemos que asociarle el atributo `ng-options`:

```
<select ng-model="user.province" required ng-options="province.code as
 province.name for province in provinces">
  <option value="">-- Seleccione una opción --</option>
</select>
```

En el ejemplo, estamos iterarndo el array de provincias. El valor que se asocia al modelo es el código de provincia. Sin embargo, la opción que se muestra es el nombre de dicha provincia. Además, establecemos un valor vacío por defecto, con `option value=""`

Aunque esta es la forma más habitual de trabajar con un `select` en AngularJS, éste nos permite hacerlo de muchas más maneras. La ayuda de AngularJS[54], nos explica cómo hacerlo de todas las maneras posibles cuando esta fuente es un array de cadenas, o un array de objetos.

[54] https://docs.angularjs.org/api/ng/directive/select

Checkboxes

Un `checkbox` no ex más que un valor booleano. En nuestro formulario, la directiva `input` le asociará el valor `true` o `false` al modelo en función de si está marcado o no. En caso de estar marcado, suscribiremos a nuestro usuario al boletín de noticias.

```
<input type="checkbox" ng-model="user.newsletter" />
```

Probando el formulario con las nuevas restricciones

Si probamos ahora el formulario[55], con las restricciones de validación que hemos añadido, veremos en el elemento `<pre>` que el objeto adquiere valores cuando superamos dichas restricciones.

8.4. Mejorando la experiencia mobile

Hemos visto el uso de ciertas directivas para validación de URLs, fechas o emails que cambiaban el atributo `type` de nuestros `inputs`. Esto tiene una ventaja adicional: usar estos tipos mejoran la experiencia de uso cuando empleamos dispositivos móviles, ya que permiten que el usuario se evite presionar varias veces el teclado para pulsar botones que debería tener a mano. Esto se debe a que el layout del teclado de nuestros móviles se adapta al tipo de `input` que estamos definiendo. Y, si estamos usando AngularJS, tenemos la ventaja que que la validación del tipo de dato está garantizada.

Ya hemos visto algunos, pero hay más que merece la pena conocer. Repasémoslos todos.

text

El tipo estándar que ya conocemos todos.

[55] http://codepen.io/alexsuch/pen/yKbre

email

Muchas veces habremos visto lo incómodo que es introducir un email en nuestro móvil, porque la `@` siempre está oculta. Esto se soluciona con el `type="email"`, ya que la hace visible. En muchos casos, además, hace que el teclado muestre directamente un botón `.com`, ya que es la extensión más habitual.

tel

El `type="tel"` abre un teclado numérico, permitiendo al usuario introducir un número de teléfono, y los caracteres típicos asociados a los teléfonos.

number

Nos permite introducir números y símbolos.

password

Conocido por todos, oculta los caracteres de una contraseña de la vista de curiosos.

date

Ya no nos tendremos que preocupar en nuestros móviles de componentes de tipo calendario, ya que el `type="date"` nos muestra, en el teclado nativo de nuestro dispositivo, un selector de fechas muy cómodo de utilizar.

month

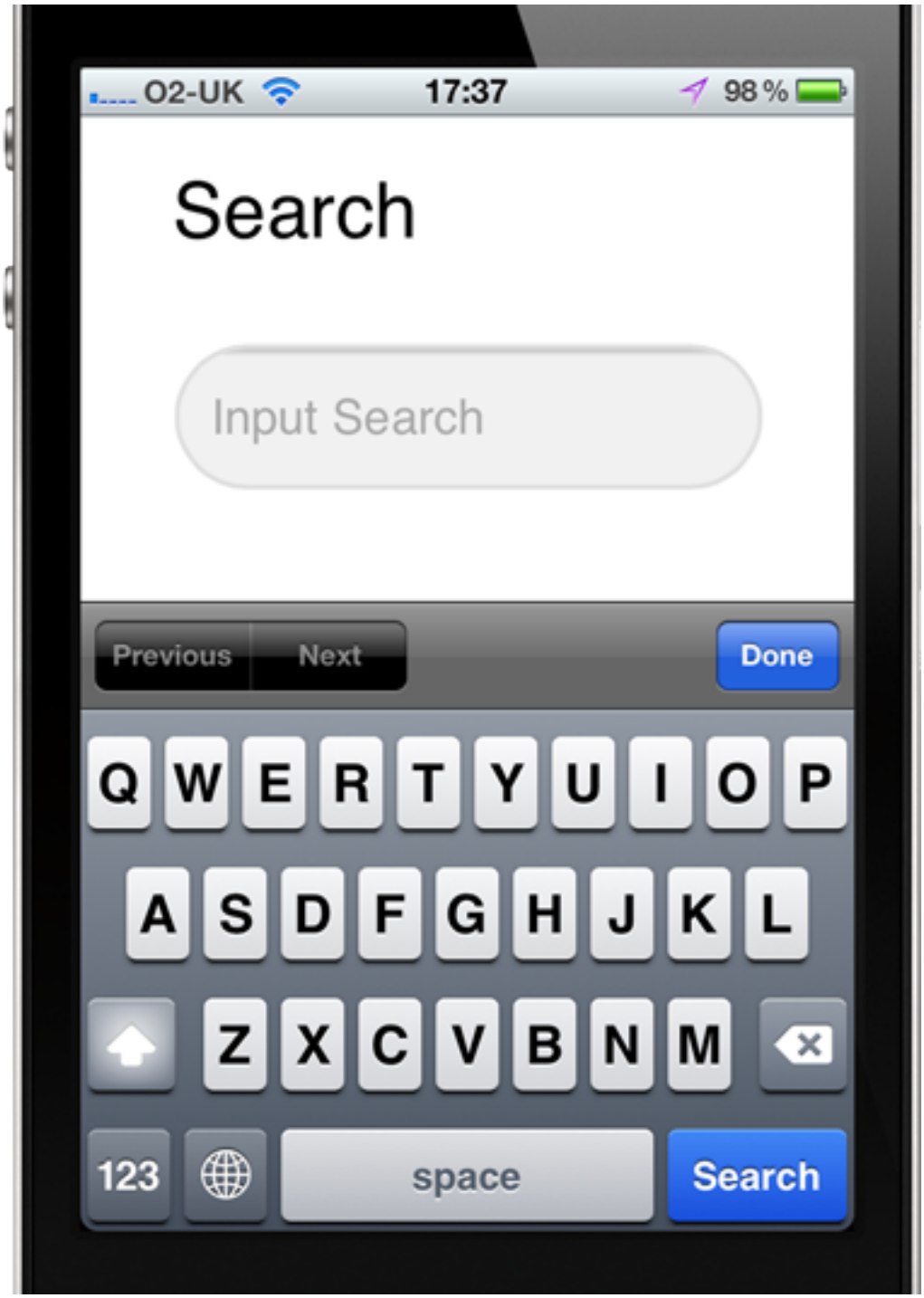
El `type="month"` es similar al `date`, permitiéndonos seleccionar un mes y un año.

datetime

Otro selector de fechas, esta vez más completo ya que el `type="datetime"` nos permite seleccionar una fecha y una hora.

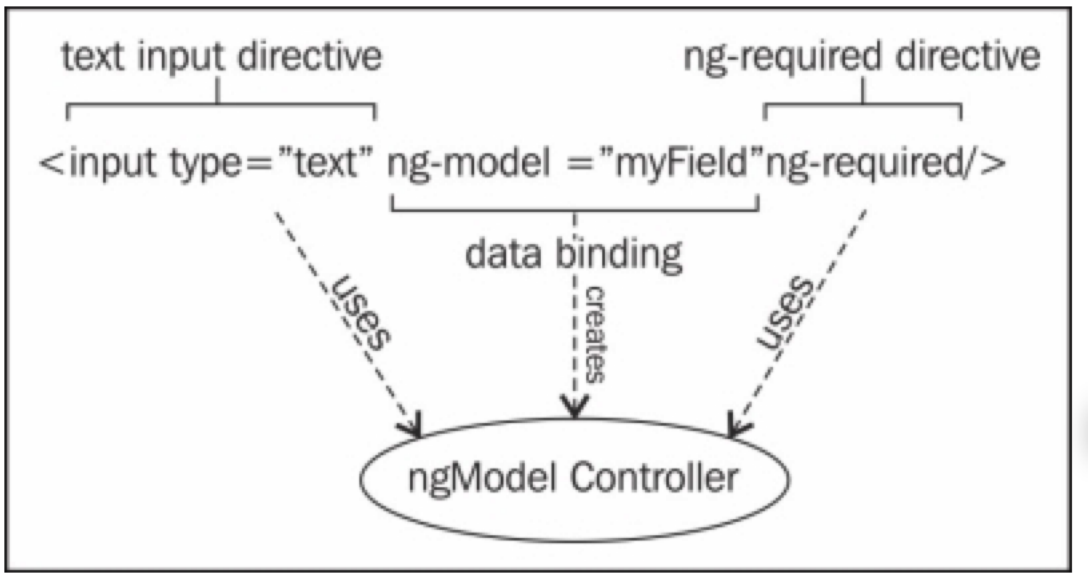
search

El `input type="search"` reemplaza el botón ***ok*** de nuestros teclados por un botón ***buscar***.



8.5. El controlador `ngModelController`

Cada directiva `ngModel` crea una instancia de `ngModelController`. Se trata de un controlador que estará disponible en todas las directivas asociadas al elemento `input`



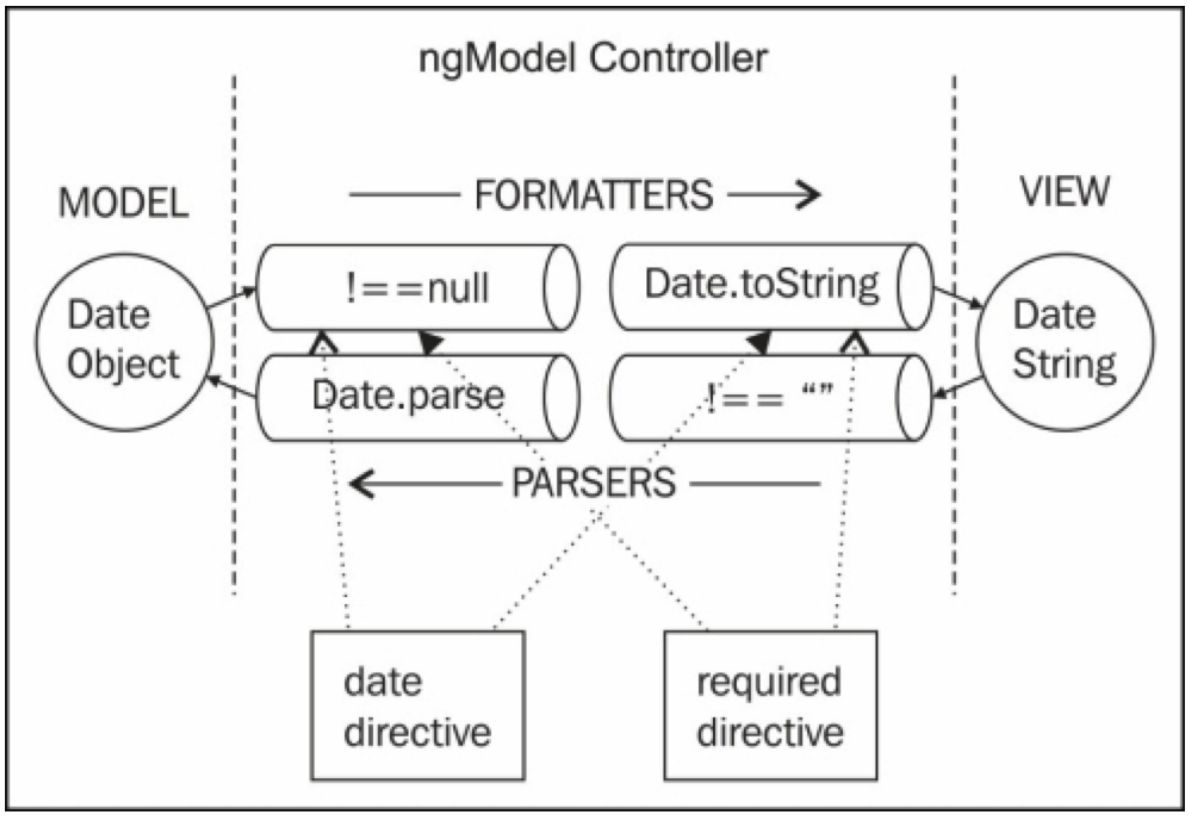
El controlador `ngModelController` es el encargado de gestionar el *data binding* entre el valor almacenado en la el modelo, y el que se muestra en el elemento `input`.

Además, el `ngModelController` se encarga de determinar que el valor de la vista es válido, y si el `input` lo ha modificado para actualizar el modelo.

Para esta actualización, sigue un *pipeline* de transformaciones que se producen cada vez que se actualiza el *data binding*. Este *pipeline* consiste en dos arrays:

- `$formatters`: transforman el dato del modelo a la vista. Tengamos en cuenta que los inputs sólo entienden datos de tipo texto, mientras que los datos en el modelo pueden ser objetos complejos.
- `$parsers`: transforman los datos de la vista a objetos del modelo.

Cualquier directiva que creemos, puede añadir sus propios `parsers` y `formatters` al *pipeline* para modificar lo que ocurre en el *data binding*. En la siguiente imagen podemos ver cómo afecta el uso de las directivas `date` y `required`. La directiva `date` parsea y formatea las fechas, mientras que la directiva `required` se asegura que no falte el valor.



Seguimiento de cambios en el modelo

Además de transformar el valor entre el modelo y la vista el `ngModelController` realiza un seguimiento de cambios.

Cuando se inicializa por primera vez, el `ngModelController` marca el valor como `pristine` (limpio, no modificado). Además, añade al `input` la clase CSS `.ng-pristine`. Una vez cambia el valor en la vista, se marca como `dirty`, y la clase `.ng-pristine` se substituye por `.ng-dirty`.

Gracias a estos estilos CSS, podemos cambiar la apariencia de nuestros elementos `input` en función de si el usuario ha introducido datos o no.

Las siguientes reglas de CSS hacen el elemento más grueso cuando introducimos datos en un input:

```
.ng-pristine { border: 1px solid black ; }

.ng-dirty { border: 3px solid black; }
```

Seguimiento de la validez del dato

Al igual que podemos realizar un *tracking* de cambios sobre el modelo, también lo podemos hacer sobre un dato válido o no.

De manera análoga a como hacía para los valores modificados o no, el `ngModelController` introduce las clases CSS `.ng-valid` y `.ng-invalid` cuando la validación de un elemento es correcta o no.

Por ejemplo, para marcar de verde o rojo los elementos modificados en función de su validez, utilizaremos las siguientes reglas CSS:

```
.ng-valid.ng-dirty {
  border: 3px solid green;
}

.ng-invalid.ng-dirty {
  border: 3px solid red;
}
```

8.6. El controlador `ngFormController`

Al igual que cada `ng-model` genera un `ngModelController`, todo elemento `form` genera un controlador `ngFormController`. Éste hace uso de todos los `ngModelController` en su interior y determina si el formulario está `pristine` o `dirty`, así como `valid` o `invalid`.

Esto es posible porque, cuando se crea un `ngModelController`, éste busca un formulario en el árbol del DOM, y se registra en el primero que encuentra. Así, el `ngFormController` sabe a qué directivas debe realizar un seguimiento.

Dando nombres a los elementos

Podemos conseguir que el `ngFormController` aparezca en el scope, simplemente dándole un nombre al formularios. Además, si damos nombre a todos los elementos `input` que tengan una directiva `ngModelController`, éstos aparecerán como propiedades del objeto `ngModelController`.

```
<form ng-submit="submitAction()" name="userForm">
```

```
<input type="email" ng-model="user.email" required name="userEmail" />
```

Validación programática

Al tener los objetos `ngModelController` y `ngFormController` en el `scope`, podemos trabajar con el estado del formulario de maner programática, usando los valores `$dirty` e `$invalid` para cambiar lo que está habilitado o visible para el usuario.

Por ejemplo, podemos hacer uso de la directiva `ng-class`[56] para mostrar los elementos que no son válidos:

```
.invalidelement { border: 1px solid #f00; }
.validelement { border: 1px solid #0f0; }
```

Aunque podemos hacerlo directamente en la vista:

```
<label>
```

[56] https://docs.angularjs.org/api/ng/directive/ngClass

```
  nombre:
  <input type="text" ng-model="user.name" required ng-minlength="3" ng-
maxlength="25" name="userName" ng-class="{ 'invalidelement' :
 userForm.userName.$invalid, 'validelement' : userForm.userName.$valid }"
 />
```

Es más adecuado y duplicamos menos código llevando esta funcionalidad al controlador:

```
$scope.getCssClasses  = function(ngModelCtrl){
  return {
    invalidelement: ngModelCtrl.$invalid && ngModelCtrl.$dirty,
    validelement: ngModelCtrl.$valid && ngModelCtrl.$dirty
  };
};
```

```
<label>
  nombre:
  <input type="text" ng-model="user.name" required ng-minlength="3" ng-
maxlength="25" name="userName" ng-class="getCssClasses(userForm.userName)"
 />
</label>
```

Para mostrar los errores de validación, haremos uso de la directiva `ngShow` [57]:

```
$scope.showError = function(ngModelCtrl, error) {
  return ngModelCtrl.$error[error];
};
```

```
  <div>
      <label>
        nombre: <input type="text" ng-model="user.name" required ng-
minlength="3" ng-maxlength="25" name="userName" ng-
class="getCssClasses(userForm.userName)" />
      </label>
      <span ng-show="showError(userForm.userName, 'required')">
        Campo obligatorio
      </span>
      <span ng-show="showError(userForm.userName, 'minlength')">
        Longitud mínima: 3
      </span>
      <span ng-show="showError(userForm.userName, 'maxlength')">
        Longitud máxima: 25
      </span>
    </div>
```

En http://codepen.io/alexsuch/pen/fJgaK tenemos un ejemplo funcionando con todas las validaciones y comprobaciones del formulario. Además, en él también hemos introducido un botón para enviar el formulario. Sin embargo no nos interesará enviarlo a menos que estén

[57] https://docs.angularjs.org/api/ng/directive/ngShow

todos los campos correctamente introducidos. Es por ello que haremos uso de la directiva `ngDisabled` [58] para deshabilitar el botón si el formulario no es válido.

```
<button type="submit" ng-disabled="userForm.$invalid">Registrar</button>
```

8.7. Ejercicio (0.5 puntos)

Aplica el *tag* `form` a la versión que quieres que se corrija.

Crea una nueva ruta en nuestra página del carrito, llamada `/edit/:productId`, donde mostraremos un formulario donde editar nuestro producto. Tendrá los campos:

- Marca. Texto obligatorio. Longitud máxima: 55 caracteres.
- Modelo. Texto obligatorio. Longitud máxima: 255 caracteres.
- Precio. Número obligatorio. Mínimo: 0. Máximo: 999.
- Descripción: Texto obligatorio. Debe contener al menos dos palabras y terminar en punto. Utilizaremos expresiones regulares para validarlo.

El formulario tendrá un botón *Guardar*, que estará deshabilitado mientras algún ítem del formulario sea incorrecto. Cuando se válido, se añadirá el ítem al listado de productos. <<<

9. Custom directives

A lo largo de los distintos capítulos hemos visto que casi todo lo que utilizamos en nuestras plantillas HTML es una directiva. Las directivas son los elementos que nos permiten extender el DOM, generando componentes con el comportamiento que nosotros queramos.

Aunque AngularJS trae un conjunto de directivas muy potente en su *core*, en alguna ocasion querremos crear elementos con cierta funcionalidad propia y reusable. En este capítulo veremos cómo podemos hacerlo a través de la creación de nuevas directivas.

9.1. Definiendo nuevas directivas

Podemos definir nuevas directivas gracias al método `directive()`, que nos proporciona un `module` de AngularJS. La definición es similar a como ya hemos visto para controladores, filtros o servicios. El nombre de la directiva debe definirse en *camelCase*, y la función debe devolver un objeto, conocido como *Directive Definition Object* (DDO).

> En nuestro código javascript, definiremos nuestra directiva en *camelCase* (ej: `youtubeVideo`). Sin embargo en nuestras plantillas tendremos cada palabra separada por guiones (ej: `youtubeVideo` pasa a convertirse en `youtube-video`)

9.2. Nuestra primera directiva

Vamos a crear una directiva muy sencilla, cuya etiqueta será `login-button`. En ella, haciendo clic en el botón de login, seríamos redirigidos a la página de login de nuestra aplicación:

```
<div ng-app="directives1">
```

[58] https://docs.angularjs.org/api/ng/directive/ngDisabled

```
  <login-button></login-button>
</div>
```

Al introducir esta directiva en nuestro módulo, AngularJS compilará el HTML e invocará esta directiva. El **DDO** de la directiva es:

```
angular
 .module('directives1', [])
 .directive('loginButton', function(){
  return {
   restrict : 'E',
   template : '<a class="btn btn-primary btn-lg" ng-href="#/login"><span
 class="glyphicon glyphicon-log-in"></span> Acceder</a>'
  }
 });
```

Inspeccionando el código de la aplicación[59], vemos que el contenido de la etiqueta se ha reemplazado por el atributo `template` de nuestro DDO.

```
<body>
 <div ng-app="directives1" class="ng-scope">
  <login-button>
   <a class="btn btn-primary btn-lg" ng-href="#/login" href="#/login">
    <span class="glyphicon glyphicon-log-in"></span>
    Acceder
   </a>
  </login-button>
 </div>
</body>
```

9.3. El atributo `restrict`

Hemos visto que la directiva `loginButton` consiste en una etiqueta HTML. Esto se debe al valor del atributo `restrict`. Éste acepta los siguientes valores:

- `E` para elementos: `<login-button></login-button>`.
- `A` para atributos: `<span login-button></span>`.
- `C` para clases: `<div class="login-button"></div>`.
- `M` para comentarios: `<!-- directive: login-button -->`.

Pero una directiva no tiene por qué ser de un tipo únicamente. Podemos definir varios tipos el atributo `restrict`. En el siguiente ejemplo[60], nuestra directiva será capaz de funcionar como atributo, elemento o clase.

```
restrict : 'EAC'
```

```
<login-button></login-button>
```

[59] http://codepen.io/alexsuch/pen/nyuGp
[60] http://codepen.io/alexsuch/pen/kjDFL

```
<div login-button></div>

<div class="login-button"></div>
```

Aunque disponemos de estas cuatro maneras de crear directivas, la declaración que mejor compatibilidad tienen con todos los navegadores es `A` (atributo).

Como os habréis imaginado, cuando hablamos de problemas con navegadores estamos haciendo una referencia indirecta a Internet Explorer. Aquí[61] hay cierta información de los problemas de compatibilidad de las directivas con algunas versiones de Internet explorer y cómo solventarlas.

9.4. Paso de datos a la directiva

Nuestra directiva `loginButton` está muy bien. Sin embargo, no todas las aplicaciones tienen el acceso en la ruta `#/login`. Además, si quisiéramos internacionalizar nuestra aplicación, tampoco sería recomendable poner la palabra *Acceso* allí donde la hemos puesto.

Quizá sería mejor refactorizar nuestra directiva para que soporte estas posibilidades, y pasarle estos datos como atributos de la siguiente manera:

```
<div login-button
 login-path="#/login"
 login-text="Área de usuario">
</div>
```

Aunque podríamos haber cogido estos datos directamente del `$scope` o `$rootScope`, esto puede acarrear problemas si el dato se elimina. Para solucionar esto, Angular nos permite crear un *scope* hijo, o lo que se conoce como un *isolate scope*. Este segundo está completamente separado del *scope* padre en el DOM, y se crea de una manera sencilla: simplemente definiremos un atributo `scope` en nuestro DDO[62]:

```
angular
 .module('directives3', [])
 .directive('loginButton', function(){
  return {
   restrict : 'A',

   scope: {
    loginPath : '@',
    loginText : '@'
   },

   template : '<a class="btn btn-primary btn-lg" ng-
href="{{loginPath}}"><span class="glyphicon glyphicon-log-in"></span>
 {{loginText}}</a>'
  }
 });
```

[61] https://docs.angularjs.org/guide/ie
[62] http://codepen.io/alexsuch/pen/cdDJA

Vemos cómo hemos introducido un objeto `scope`, y además hemos modificado el `template` para hacer *binding* con las variables definidas en él.

> Vemos que la convención camelCase se mantiene también para los atributos del *scope*.

La convención de nombrado por defectoes que el atributo y la propiedad del `scope` se llamen igual. En algunas ocasiones podríamos querer que la variable del scope tuviese un nombre distinto. Para ello especificaríamos los nombres de la siguiente manera:

```
scope : {
 loginPath : '@uri',
 loginText : '@customText'
}
```

y, en nuestra plantilla:

```
<div login-button
 uri="#/login"
 custom-text="Área de usuario">
</div>
```

Aquí, estamos diciendo que se establezca el valor de la variable `loginPath` del *isolate scope* con lo que pasamos como atributo `uri`.

Ahora, imaginemos que no queremos tener la URL *hardcodeada*, ya que tenemos un servicio en nuestra aplicación que nos proporciona todas las URLs de la misma. Nuestro controlador pasa dicho servicio a la vista:

```
.controller('MainCtrl', function($scope, AppUrls){
  $scope.urls = AppUrls;
 })
```

y la plantilla pasa la url como parámetro de la directiva:

```
<div login-button
 login-path="urls.login"
 login-text="Área de usuario">
</div>
```

Tras este cambio, si observamos el fuente de nuestra aplicación, veremos que no tenemos el resultado que esperábamos, y en lugar de un esperado `a href="#/login"`, nuestro enlace es `a href="urls.login"`.

Para obtener el resultado esperado, tenemos que hacer una ligera modificación en el `scope` de nuestra directiva:

```
scope : {
 loginPath : '=',
```

```
  loginText : '@'
}
```

Podemos ver un ejemplo completo de este funcionamiento en http://codepen.io/alexsuch/pen/mdAFa.

Vemos que hemos cambiado la primera `@` por un `=`. Este símbolo determina la **estrategia de binding**:

- `@` lee el valor de un atributo. El valor final siempre será una cadena. Al leerse tras la evaluación del DOM, también podemos usarlo de la forma `title="{{title}}"`, el valor del atributo es el que hayamos establecido en el `$scope` para la variable `title`.
- `=` nos permite realizar el *two-way data binding* en nuestra directiva, *bindando* la propiedad del `scope` de la directiva a una propiedad del `scope` padre. Cuando utilicemos `=`, usaremos el nombre de la propiedad sin las llaves `{{}}`.
- `&` permite realizar referencias a funciones en el `scope` padre.

9.5. El atributo `transclude`

Vamos a realizar una modificación adicional en nuestro botón, haciendo que se parezca más a una etiqueta HTML. En un enlace normal introduciríamos el texto detro de la etiqueta, en lugar de como un atributo, ¿no?. Vamos a hacer lo mismo, con nuestra directiva, para que tenga la forma:

```
<div login-button login-path="urls.login">
  Área de usuario
</div>
```

Para ello, haremos uso de la *transclusión*. Ésta consiste en la inclusión de un documento (o parte de un documento) dentro de otro documento. Realmente es algo que ya hemos visto en la parte de *routing* con las directivas `ngView` o `uiView`, donde introducíamos unas plantillas dentro de otras.

Es justo lo que vamos a hacer en nuestra directiva: queremos que el fragmento *Área de usuario* se introduzca en una zona concreta de la plantilla.

Para permitir la *transclusión* en AngularJS, debemos hacer dos cosas:

En primer lugar, nuestra directiva tiene que permitir la *transclusión*. Para ello, añadir el atributo `transclusion` al DDO.

Además, tenemos que indicar dónde se va a realizar la *transclusión*. AngularJS nos proporciona la directiva `ngTransclude`, que introduciremos en la parte de nuestra plantilla dode queramos realizarla.

Así, nuestro DDO queda de la siguiente manera:

```
{
 restrict : 'A',

 scope : {
  loginPath : '='
 },
```

```
  transclude : true,

  template : '<a class="btn btn-primary btn-lg" ng-
href="{{loginPath}}"><span class="glyphicon glyphicon-log-in"></span>
  <span ng-transclude></span></a>'
}
```

Ahora, ya no es necesario el atributo `loginText`, y por eso se ha eliminado. Se ha substituído por un `<span ng-transclude></span>` en la plantilla.

Podemos ver este ejemplo funcionando en http://codepen.io/alexsuch/pen/DxjEu.

9.6. Un vistazo a todas las propiedades de una directiva

Ahora que tenemos un varios ejemplos de cómo crear una directiva, veamos cuáles son todas las propiedades que podemos definir en una directiva:

restrict

Como hemos visto, permite determinar cómo puede usarse una directiva:

- `A`tributo
- `E`lemento
- `C`lase
- Co`M`entario

scope

Lo utlizamos para crear un *scope* hijo (`scope : true`) o un *isolate scope* (`scope : {}`).

template

Define el contenido de la directiva. Puede incluir código HTML, *data binding expressions* y otras directivas.

templateUrl

Al igual que en el *routing*, podemos definir un path para la plantilla de nuestra directiva.

Definir un `templateUrl` puede ser útil en componentes muy específicos para una aplicación. Sin embargo, cuando desarrollamos componentes reutilizables, lo mejor es definir la plantilla dentro de la directiva como un atributo `template`.

controller

Nos permite definir un controlador, que se asociará a la plantilla de la directiva de igual manera que hacíamos en el *routing*.

Recibe como parámetros cuatro argumentos:

```
angular
  .module('exampleModule', [])
  .directive('exampleDirective', function(){
```

```
  return {
   restrict : 'A',
   controller : function($scope, $element, $attrs, $transclude) {
    //Código de nuestro controlador
   }
  };
 })
```

`$scope`

Hace referencia al objeto `scope` asociado a la directiva.

`$element`

Hace referencia al objeto jqLite (similiar a un objeto jQuery) de la directiva.

=====`$attrs` Hace referencia a los atributos del elemento. Por ejemplo para un elemento

```
<div id="myId" class="blue-bordered"></div>
```

el objeto `$attrs` tendría el valor:

```
{
 id : 'myId',
 class : 'blue-bordered'
}
```

$transclude

Esta función crea un clon del elemento a transcluir, permitiéndonos manipular el DOM.

En teoría, aunque podemos manipular el DOM desde un controlador, el lugar adecuado donde hacerlo es en el código de una directiva.

El siguiente ejemplo crea un enlace vacío con el texto a transcluir a continuación de nuestro elemento:

```
controller : function($scope, $element, $attrs,$transclude) {
 $transclude(function(clone){
  console.log('clone is', clone)
  console.log('clone txt is', clone.text())

  var a = angular.element('<span>');
  a.text(clone.text());
  $element.append(a);
 });
},
```

transclude

Nos permite realizar la *transclusión* del bloque HTML que queramos, combinando su uso con la directiva `ngTransclude`.

replace

Si inspeccionamos el código de nuestra aplicación, veremos que cuando introducimos una directiva se crea un elemento padre con la definición de la directiva, y dentro de él se desarrolla la directiva.

Cuando se establece con el valor `true`, reemplazamos el elemento padre por el valor de la directiva, en lugar de introducirlo como hijo.

Es decir, pasamos de

```
<div ng-app="directives5" ng-controller="MainCtrl" class="ng-scope">
 <div login-button="" login-path="urls.login" class="ng-isolate-scope">
  <a class="btn btn-primary btn-lg" ng-href="#/login" href="#/login">
   <span class="glyphicon glyphicon-log-in"></span>
   <span ng-transclude="">
    <span class="ng-scope">
     Área de usuario
    </span>
   </span>
  </a>
 </div>
</div>
```

a

```
<div ng-app="directives5" ng-controller="MainCtrl" class="ng-scope">
 <a class="btn btn-primary btn-lg" ng-href="#/login" href="#/login">
  <span class="glyphicon glyphicon-log-in"></span>
  <span ng-transclude="">
   <span class="ng-scope">
    Área de usuario
   </span>
  </span>
 </a>
</div>
```

Vemos que, en el segundo caso, ha desaparecido el bloque `div login-button`.

link

El `template` o `templateUrl` no tiene utilidad hasta que se compila contra un `scope`. Hemos visto que una directiva no tiene un `scope` por defecto, y utilizará el del padre a no ser que se lo indiquemos.

Para hacer uso del scope, utilizaremos la función `link`, que recibe tres argumentos:

- `scope`: el `scope` que se pasa a la directiva, pudiendo ser propio o el del padre.
- `element`: un elemento jQLite (un *subset* de jQuery) donde se aplica nuestra directiva. Si tenemos jQuery instalado en nuestra aplicación, será un elemento jQuery en lugar de un lQLite.
- `attrs`: un objeto que contiene todos los atributos del elemento donde aplicamos nuestra directiva, de igual manera que vimos con el controlador.

El uso principal de la función `link` es para asociar *listeners* a elementos del DOM, observar cambios en propiedades del modelo, y validación de elementos.

require

La opción `require` puede ser una cadena array de cadenas, correspondoentes a nombres de directivas. Al usarla, se asume que esas directivas indicadas en el array han sido previamente aplicadas en el propio elemento, o en su elemento padre (si se ha marcado con un `^`). Se utiliza para inyectar el controlador de la directiva requerida como cuarto argumento de la función `link` de nuestra directiva.

Esta cadena o conjunto de cadenas se corresponde con el nombre de las directivas cuyo controlador queremos utilizar.

```
// ...
  restrict : 'EA',
  require : 'ngModel' // el elemento debe tener la directiva ngModel para
 poder utilizar su controlador
// ...
```

```
// ...
  restrict : 'EA',
  require : '^ngModel' // el elemento o su padre, deben tener la directiva
 ngModel
// ...
```

compile

Utilizaremos la función `compile` para realizar transformaciones en el DOM antes de que se ejecute la función `link`. Esta función recibe dos elementos:

- `element`: elemento sobre el que se aplicará la directiva
- `attrs`: listado de atributos de la directiva.

La función `compile` no tiene acceso al `scope`, y debe devolver una función `link`.

El esqueleto de una directiva cuando utilizamos una función `compile` es:

```
angular
  .module('compileSkel', [])
  .directive('sample', function(){
    return {
      compile : function(element, attrs) {
        //realizar transformaciones sobre el DOM
        return function(scope, element, attrs){
          //función link normal y corriente
        };
      }
    };
  });
```

En http://codepen.io/alexsuch/pen/LpcsK tenemos un ejemplo que usa la función `compile` para establecer un estilo por defecto a una serie de elementos `div`.

9.7. Directivas de validación

Éste es un buen ejemplo para varias cosas. Por un lado, veremos un ejemplo real del funcionamiento del atributo `require` en una directiva. Además, haremos uso del controlador ngModelController que habíamos visto en la parte de los formularios. Por último, veremos cómo ampliar la API de validadción de AngularJS con directivas propias, ya que ésta es la manera recomendada de implementar la validación en AngularJS.

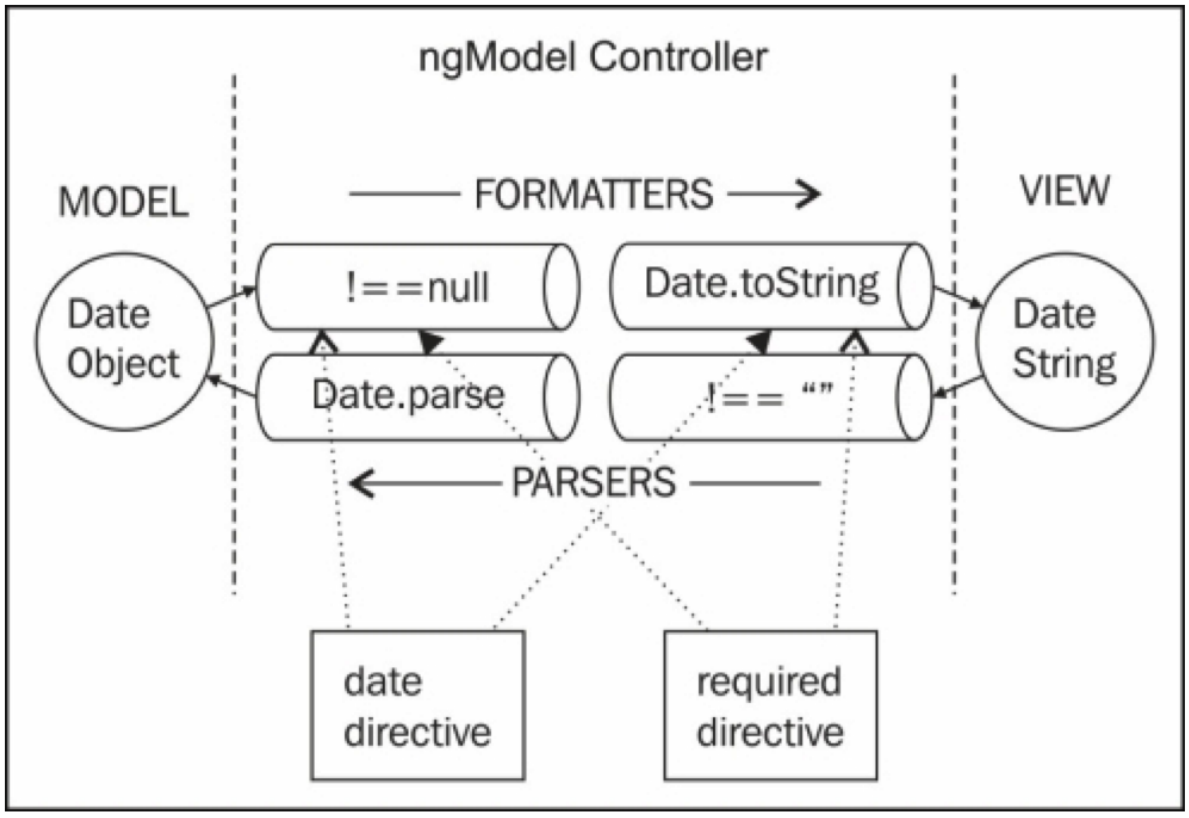
Para el ejemplo, introduciremos una directiva que valide DNIs. Como todos sabremos, la letra del DNI es un dígito de control que se obtiene al aplicar una fórmula matemática sobre el número. Nuestra directiva validará que la longitud del DNI y el dígito de control sean correctos. Tendrá una forma similar a:

```
<input type="text" ng-model="user.dni" is-dni />
```

Internet está lleno de sitios donde encontrar la fórmula de validación del DNI. Nosotros la hemos obtenido de aquí[63], porque también acepta NIEs.

Para este tipo de directivas, vamos a tener que hacer, sí o sí, una restricción de tipo atributo.

También, recordemos cómo funcionaba el *pipeline* de *parsers* y *formatters* que vimos en el capítulo anterior:



[63] http://www.yporqueno.es/blog/javascript-validar-dni

Lo que haremos en nuestra directiva será introducir nuestro validador como primer elemento del array de *parsers*, y como último elemento del array de *formatters* para el `ngModelController`.

```
angular
  .module('validator.nif', [])
  .directive('isNif', function () {
    return {
      restrict: 'A',
      require: 'ngModel',
      link: function(scope, element, attrs, ngModelCtrl) {
        // Acepta NIEs (Extranjeros con X, Y o Z al principio)
        // http://www.yporqueno.es/blog/javascript-validar-dni
        var validarNif = function (value) {
          var numero, letraDni, letra;
          var expresion_regular_dni = /^[XYZ]?\d{5,8}[A-Z]$/;
          var result;

          value = value.toUpperCase();

          if (expresion_regular_dni.test(value) === true) {
            numero = value.substr(0, value.length - 1);
            numero = numero.replace('X', 0);
            numero = numero.replace('Y', 1);
            numero = numero.replace('Z', 2);
            letraDni = value.substr(value.length - 1, 1);
            numero = numero % 23;
            letra = 'TRWAGMYFPDXBNJZSQVHLCKET';
            letra = letra.substring(numero, numero + 1);

            if (letra != letraDni) {
              result = false;
            } else {
              result = true;
            }
          } else {
            result = false;
          }

          ngModelCtrl.$setValidity('isNif', result);
          return value;
        };

        ngModelCtrl.$parsers.unshift(validarNif);
        ngModelCtrl.$formatters.push(validarNif);
      }
    }
  });
```

Como podemos ver en la función `validarNif`, el validador tiene una doble responsabilidad:

- Indicar al `ngModelController` si la validación de DNI/NIE es correcta o no.
- Devolver el nuevo valor del modelo. En nuestro caso estamos devolviendo siempre el valor del modelo, pero puede haber circunstancias en que queramos devolver otra cosa.

También, en el caso de los *formatters* podríamos querer transformar la cadena a objeto antes de dárselo al controlador.

Podemos ver un ejemplo de nuestra directiva funcionando en http://codepen.io/alexsuch/pen/cdzyw.

9.8. Ejercicios (2 puntos)

Aplica el *tag* `directives` a la versión que quieres que se corrija.

Directiva de componente (1 puntos)

Vamos a crear una directiva llamada `product`, que requiere un `ngModel`. Como *template* tendrá el div que habíamos hecho para pintar los productos del carrito. Aquí está nuestra plantilla de directiva:

```
angular
  .module('org.expertojava.carrito')
  .directive('product', function(){
    return {
      require: '', //¿requiere algo?
      restrict: '', //¿qué restrict le ponemos?
      scope: {
        product: '' //ver qué ponemos aquí.
      },
      templateUrl: '' //crear plantilla
    };
  })
```

Cambiaremos el bucle que mostraba los productos por el siguiente:

```
<div ng-repeat="product in products">
  <product ng-model="product" />
</div>
```

o bien por este otro:

```
<div ng-repeat="product in products">
  <div product ng-model="product"></div>
</div>
```

Directiva de validación (1 puntos)

Vamos a crear una directiva, llamada `maxDecimals`, que usaremos en el formulario de edición del capítulo anterior. El dato será inválido si el número de decimales del input es mayor que valor que se asigne a `max-decimals`. Ejemplo:

```
<input type="number" required max-decimals="2" />
```

La plantilla de nuestra directiva será:

```
angular
 .module('org.expertojava.carrito')
 .directive('maxDecimals', function(){
   return {
     require: 'ngModel',
     restrict: '', //TODO: ¿Qué ponemos aquí?
     link: function(scope, element, attrs, ngModelCtrl) {
       var validationFn = function(value){
         var maxDecimals = attrs.maxDecimals;
         var validity = true;

         //TODO: IMPLEMENTAR LÓGICA

         ngModelCtrl.$setValidity('maxDecimals', validity);

         return value;
       };

       //TODO: ASOCIAR A PARSERS Y FORMATTERS
     }
   };
 });
```

10. Promesas de resultados

Una promesa, o *deferred object*, es una herramienta muy sencilla y útil cuando realizamos programación asíncrona.

Aunque hay muchas implementaciones en JavaScript, el equipo de AngularJS realizó una adaptación de la librería Q, de Kris Kowal[64], debido a su éxito y difusión.

10.1. Por qué utilizar promesas

En JavaScript, los métodos asíncronos normalmente utilizan funciones de *callback* para devolver un estado de éxito o error. Por ejemplo, la API de gelocalización requiere de estas dos funciones de *callback* para obtener la posición actual[65]:

```
var successFn = function(response){
 console.log('SUCCESS! ' + JSON.stringify(response));
};

var errorFn = function(err) {
  console.log('ERROR! ' + JSON.stringify(response));
}

navigator.geolocation.getCurrentPosition(successFn, errorFn);
```

Otro ejemplo es el objeto XMLHttpRequest, que utilizamos para realizar peticiones AJAX. Tiene una función de *callback* llamada `onReadyStateChange`, que se llama cuando cambia el atributo `readyState`.

```
var xhr = new window.XMLHttpRequest();
xhr.open('GET', 'http://www.webdeveasy.com', true);
xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) {
        if (xhr.status === 200) {
            console.log('Success');
        }
    }
};
xhr.send();
```

En nuestro día a día nos encontraremos con una infinidad de usos y ejemplos. Aunque puede parecer sencillo de manejar, puede volverse un infierno cuando tenemos que encadenar funciones de sincronización.

64 https://github.com/kriskowal/q
65 http://codepen.io/alexsuch/pen/xgzCn

```
// get all the original jpegs, and create the edited jpegs.
function reserveWork()
{
  beanstalkd.reserve(function(err, jobid, payload){ reportError(err);
    beanstalkd.bury(jobid, 1024, function(err){ reportError(err);
      var spin = JSON.parse(payload.toString());
      console.dir(spin);
      spin.shortid = spin.short_id;
      var s3key = spin.shortid+"/spin.zip";

      console.log(("["+spin.shortid+"] STARTED (beanjob #"+jobid+")"));


      s3.getObject({Bucket: s3bucket, Key: s3key}, function(err, data) { if(err) console.error("Could not get "+s3key); reportError(err);
        fs.mkdirs(path.dirname(s3key),function(err){ reportError(err);
          fs.writeFile(s3key,data.Body,function(err){ reportError(err);
            // spin.zip is on the file system now
            var cmd = "unzip -o "+s3key+" -d "+path.dirname(s3key);
            exec(cmd,function(){
              console.log("Stuff is unzipped!");

              fs.mkdirs(path.dirname(s3key)+"/orig",function(){
                var vfs = ["null"];
                var rots = [null, "transpose=2", "transpose=2,transpose=2", "transpose=2,transpose=2,transpose=2"];
                var rotidx = parseInt(spin.rotation_angle,10)/90;
                if(rotidx) vfs.push(rots[rotidx]);
                var vf = "-vf "+vfs.join(",");
                var ffmpeg_cmd = "ffmpeg -i "+path.dirname(s3key)+"/cap.mp4 -q:v 1 "+vf+" -pix_fmt yuv420p "+path.dirname(s3key)+"/orig/%03d.jpg";
                exec(ffmpeg_cmd,function(){
                  console.log("Done with ffmpeg");
                  // Upload everything to S3
                  Step( function(){
                    for (var i=1; i<=spin.frame_count; i++)
                    {
                      var s3key = spin.shortid + "/orig/" + ("00"+i).substr(-3) + ".jpg";
                      uploadOrig(s3key, this.parallel());
                    }
                  },
                  function(){
                    fs.readFile(spin.shortid+"/labels.txt",function(err,data){
                      if(err || !data)
                        data = new Buffer("{}");
                      s3.putObject({Bucket: s3bucket, Key: spin.shortid+"/labels.json", ACL: "public-read", ContentType: "text/plain", Body: data}, function(err, data){ reportError(err);
                        console.log("All files are uploaded");
                        beanstalkd.use("editor",function(err,tube){ reportError(err,jobid);
                          beanstalkd.put(1024,0,300,JSON.stringify(spin), function(err,new_jobid){ reportError(err,jobid);
                            console.log("Added new job to beanstalkd.");
                            beanstalkd.destroy(jobid, function(){
                              console.log(("["+spin.shortid+"] FINISHED (beanjob #"+jobid+")"));
                              reserveWork();
                            });
                          });
                        });
                      });
                    });
                  });
                });

              });
            });
          });
        });
      });
    });
  });
}
```

La imagen ilustra un ejemplo de la famosa *callback pyramid of doom*. Aunque hay maneras más elegantes de escribir y refactorizar el código, siempre será difícil de leer y mantener.

10.2. Promesas y *deferred objects*

Una promesa representa el resultado de una operación asíncrona. Expone una interfaz que puede usarse para interactuar con el resultado que tendrá dicha operación. Así, también permite que quien esté interesado pueda hacer uso de dicho resultado.

La promesa está asociada a un *deferred object*, cuyo estado será "pendiente", y no tiene ningún resultado. Cuando invoquemos los métodos `resolve()` o `reject()`, este estado pasará a "resuelto" o "rechazado". Además, podemos coger la promesa una vez inicializada y definir operaciones con su resultado futuro, que se llevarán a cabo cuando se cambie a los estados "resuelto" o "rechazado" que acabamos de mencionar.

Mientras el *deferred object* tiene métodos para cambiar el estado de una operación, la promesa sólo expone métodos para operar con el resultado. Es por ello que es una buena práctica devolver una promesa y no un *deferred object*.

10.3. Promesas en AngularJS

En primer lugar, deberemos crear un *deferred object*:

```
var deferred = $q.defer();
```

El objeto `deferred` apunta a un *deferred object*, cuyo estado podremos resolver o rechazar tras realizar una operación asíncrona. Supongamos el método asíncrono

`async(successFn, errorFn)`, donde los dos parámetros son funciones de *callback*. Cuando `async` finaliza su ejecución, queremos resolver o rechazar `deferred` con su resultado:

```
async(
  function(val){
    deferred.resolve(val);
  },
  function(err){
    deferred.reject(err);
  }
);
```

Incluso podemos simplificar la llamada, ya que los métodos `resolve` y `reject` no precisan de un contexto:

```
async(deferred.resolve(val), deferred.reject(err));
```

Ahora, asignar operaciones una vez haya habido éxito o error es bastante sencillo:

```
var promise = deferred.promise;

promise
  .then(
    function(data){ alert('Success! ' + data); },
    function(data){ alert('Error! ' + data); }
  )
```

Podemos asignar tantas funciones de éxito o error como queramos. En el siguiente ejemplo, tanto las funciones asignadas antes de la llamada a `async` como las que se realizan después se ejecutarán al resultado:

```
var deferred = $q.defer();
deferred.promise
    .then(function(data) {
        console.log('Success asignado antes de invocar async()', data);
    }, function(error) {
        console.log('Error asignado antes de invocar async()', error);
    });

async(deferred.resolve, deferred.reject);

deferred.promise
    .then(function(data) {
        console.log('Success asignado tras invocar async()', data);
    }, function(error) {
        console.log('Error asignado tras invocar async()', error);
    });
```

Hemos visto que el método `then` recibe dos funciones, una de éxito y una de error. Sin embargo, podemos usar `then` para asignar funciones de éxito, y utilizar `catch` cuando se

produce un error. Además, existe una función `finally`, que se ejecutará siempre, se haya resuelto correctamente o no. Gracias a `finally`, no tendremos que duplicar código que se podría ejecutar tanto en la parte del éxito como la del error.

```
deferred.then(function(){ console.log('All OK')});

deferred.catch(function(){ console.log('Error!')});

deferred.finally(function(){ console.log('End.'); });
```

10.4. Encadenando promesas

Un dato interesante que hay que conocer, es que el método `then` de una promesa devuelve otra promesa. Cuando resolvemos la primera promesa, el valor que devolvamos se enviará a la promesa siguiente, de manera que podemos encadenar y transformar una serie de promesas de resultados. Veámoslo con un ejemplo concreto[66]:

```
function async(value) { // Supongamos que se trata de una operación
 asíncrona de verdad
    var deferred = $q.defer();
    var asyncCalculation = value / 2;
    deferred.resolve(asyncCalculation);
    return deferred.promise;
}

var promise = async(8)
    .then(function(x) {
        return x+1;
    })
    .then(function(x) {
        return x*2;
    })
    .then(function(x) {
        return x-1;
    });

promise.then(function(x) {
    console.log(x);
});
```

La promesa empieza con llamando a `async(8)`, que resuelve con el valor `4`. Este valor pasa por todas las funciones `then` secuencialmente, hasta pintar el valor `9`, ya que hace `(8 / 2 + 1) * 2 - 1`.

Como hemos visto antes que las funciones no necesitan contexto, podemos refactorizarlo[67] de la siguiente manera:

```
var promise = async(8)
                .then(addOne)
```

[66] http://codepen.io/alexsuch/pen/dlrID
[67] http://codepen.io/alexsuch/pen/jDrxu

```
                .then(mult2)
                .then(minusOne)
                .then(paintValue);
```

El ejemplo es muy optimista y asume que todo va a ir bien. Pero si no es así, ¿dónde colocamos nuestro `catch`? Bien, en caso de operaciones encadenadas, `catch` y `finally` se colocan en último lugar:

```
var promise = async(8)
                .then(addOne)
                .then(mult2)
                .then(minusOne)
                .then(paintValue)
                .catch(showError)
                .finally(endFn);
```

En el momento que cualquier elemento resuelva incorrectamente (ya sea el primero o el tercero), se ejecutará a continuación el `catch`, y se terminará con el `finally`.

10.5. Otros métodos útiles

$q.reject

En algunas ocasiones, puede que necesitemos devolver una promesa rechazada. En lugar de crear una promesa y rechazarla, podemos usar `$q.reject(reason)`, que devuelve una promesa rechazada, con el motivo que le digamos. Por ejemplo:

```
var promise = async()
  .then(
    function(value){
      if(isValid(value)) {
        return value;
      }

      return $q.reject('Valor no válido');
    }
  );
```

Si `value` es válido, un conjunto de promesas encadenadas funcionará correctamente. Sin embargo, se irá al bloque `catch` si no es válido.

$q.when

Similar a `$q.reject`, pero devuelve un valor correctamente resuelto. Un ejemplo muy claro de uso es cuando tenemos que pedir un dato al servidor si no lo tenemos cacheado.

```
function getElement(key){
  if(!!$localStorage.key) {
    return $q.when($localStorage.key);
  } else {
    return getFromServer(key);
```

```
  }
}
```

$q.all

En algunas ocasiones podríamos querer tener una serie de elementos de manera asíncrona, sin importarnos el orden, y ser notificados al terminar. Para ello, hacemos uso de `$q.all(promisesArray)`. Devuelve una promesa que se resuelve sólo cuando todas las promesas del array se han resuelto. Si al menos una de las promesas del array se rechaza, también lo hará el resultado de `$q.all`.

```
var allPromises = $q.all([
  async1(),
  async2(),
  async3(),
  ...
  asyncN()
]);

allPromises.then(function(values){
  var value1 = values[0];
  var value2 = values[1];
  var value3 = values[2];
  ...
  var valueN = values[N+1];

  console.log('end');
});
```

10.6. Ejercicio (0.5 puntos)

Genera un servicio en AngularJS que haga uso de promesas y de la API de geoposicionamiento para devolver las coordenadas del navegador. Haz también un programa en AngularJS que pinte las coordenadas en pantalla.

Aplica el *tag* `promises` a la versión que quieres que se corrija. <<<

11. Comunicación con el servidor

> Guardaremos todos los ejercicios que hagamos en la carpeta `server`
>
> Aplica el tag `server` a la versión que quieres que se corrija.
>
> Los ejercicios tienen una puntuación total de un punto, repartido equitativamente entre todos ellos.

Lo normal en una aplicación web es que, tarde o temprano, haya que comunicarse con el servidor para traernos algún tipo de dato, o bien para persistirlo. Es más, existen muchas aplicaciones que únicamente hacen CRUD, con lo que la comunicación con el servidor se convierte en algo esencial.

AngularJS dispone de una serie de APIs para comunicarse con cualquier *backend* a realizando peticiones `XMLHttpRequest` (XHR), o bien peticiones JSONP a través del servicio `$http`.

Además, existe un servicio llamado `$resource`, especializado en la comunicación con interfaces RESTful.

JSONP, o "JSON with padding", es una técinca de comunicación que usan los programas escritos en JavaScript y que corren en un navegador web. Con JSONP, podemos realizar una petición de datos a un servidor que se encuentra en otro dominio, cosa habitualmente prohibida en un navegador web debido a la *same-origin policy*.

Dado que muchos navegadores no tienen penalización *same-origin* en las etiquetas `script`, lo qu ese hace es traerse la respuesta del servidor, envuelta en una llamada a una función.

Para que JSONP funcione, el servidor al que se realizan las peticiones debe saber que tiene que devolver los resultados formateados en JSONP. Para ello, normamente se genera una URL con un parámetro llamado `callback=funcion_de_callback` (ej: http://jsonplaceholder.typicode.com/users/1?callback=processUser)

http://en.wikipedia.org/wiki/JSONP

11.1. El servicio $http

El servicio `$http` consiste en una API de propósito general para realizar peticiones XHR y JSONP. Es una API bastante sólida y sencilla de usar.

El servicio `$http` ofrece una serie de funciones que reciben como parámetros una URL y un objeto de configuración, para generar una petición HTTP. Devuelve una promesa de resultados con dos métodos: `success` y `error`.

Los métodos son equivalentes a los que podríamos hacer en una petición HTTP.

Para hacer las pruebas haremos uso de los servicios situados en JSONPlaceholder[68], que permite hacer uso del servicio `$http` sobre sus servidores, ya que tiene habilitado el soporte para CORS

$http.get

Realiza una petición `GET`, para obtener datos.

Parámetros:

- `url`: URL destino
- `config`: objeto de configuración opcional. A destacar el atributo `params`, que contiene un mapa de los parámetros a pasar.

El siguiente ejemplo[69] pide el detalle de un usuario:

```
angular
```

[68] http://jsonplaceholder.typicode.com/
[69] http://codepen.io/alexsuch/pen/wsAgj

```
  .module('httpModule', [])
  .controller('MainCtrl', function($scope, $http){
    $http.get(
      'http://jsonplaceholder.typicode.com/posts',{ params: {id:1} }
    )
    .success(function(data){
      $scope.resultdata = data;
    })
    .error(function(data){
      alert('Se ha producido un error')
    });
});
```

```
<div ng-app="httpModule" ng-controller="MainCtrl">
  <h1>Resultado</h1>
  <pre>
{{ resultdata[0] | json }}
  </pre>
  <hr />
</div>
```

EJERCICIO

Genera un pequeño programa, similar al del ejemplo, que realice una petición `GET` y devuelva el listado de comentarios para el post con ID 1.

POST

Realiza una petición `POST`, para dar de alta algún dato en el servidor.

Parámetros:

- `url`: URL destino
- `data`: datos a enviar.
- `config`: objeto de configuración opcional.

El siguiente ejemplo[70] se encarga de dar de alta un usuario:

```
angular
  .module('httpModule', [])
  .controller('MainCtrl', function($scope, $http){
    $http.post(
      'http://jsonplaceholder.typicode.com/users',
      {
          "name": "Winchester McFly",
          "username": "wmcfly",
          "email": "wmcfly@ua.es",
          "address": {
            "street": "Calle del atún 22",
          },
```

[70] http://codepen.io/alexsuch/pen/xCEeJ

```
            "phone": "666 112233",
            "website": "http://winchester-mcfly.com/"
        }
      )
      .success(function(data){
        $scope.id = data.id;
      })
      .error(function(data){
        alert('Se ha producido un error')
      });
});
```

```
<div ng-app="httpModule" ng-controller="MainCtrl">
  <h1>Resultado</h1>
  <pre ng-show="id">
Se ha dado de alta el usuario, con id: {{ id }}
  </pre>
  <hr />
</div>
```

> Podemos ver cómo falla si hacemos una petición `POST` a `/users/1`

EJERCICIO

> Genera un pequeño programa, similar al del ejemplo, que realice una petición `POST` realice el alta de una imagen.

PUT

Realiza una petición `PUT`, para actualizar algún elemento en el servidor.

Parámetros:

- `url`: URL destino
- `data`: datos a enviar
- `config`: objeto de configuración opcional.

[http://codepen.io/alexsuch/pen/FvsuH]El siguiente ejemplo se encarga de actualizar el usuario con id = 1.

```
angular
  .module('httpModule', [])
  .controller('MainCtrl', function($scope, $http){
    $http.put(
      'http://jsonplaceholder.typicode.com/users/1',
      {
          "name": "Winchester McFly",
          "username": "wmcfly",
          "email": "wmcfly@ua.es",
```

```
          "address": {
            "street": "Calle del atún 22",
          },
          "phone": "666 112233",
          "website": "http://winchester-mcfly.com/"
        }
      )
      .success(function(data){
        $scope.data = data;
      })
      .error(function(data){
        alert('Se ha producido un error')
      });
});
```

```
<div ng-app="httpModule" ng-controller="MainCtrl">
  <h1>Resultado</h1>
  <pre ng-show="data">
Se ha actualizado el usuario, sus nuevos datos son:
{{ data | json }}
  </pre>
  <hr />
</div>
```

EJERCICIO

> Genera un pequeño programa, similar al del ejemplo, que realice una petición `PUT` para actualizar el título del POST con id=1.

DELETE

Realiza una petición `DELETE`, para solicitar el borrado de algún elemento en el servidor.

Parámetros:

- `url`: URL destino
- `config`: objeto de configuración opcional.

En el siguiente ejemplo[71], eliminaremos un usuario.

```
angular
  .module('httpModule', [])
  .controller('MainCtrl', function($scope, $http){
    $http.delete(
      'http://jsonplaceholder.typicode.com/users/1')
    .success(function(data){
      alert('Se ha eliminado el usuario con éxito')
    })
    .error(function(data){
      alert('Se ha producido un error')
    });
```

[71] http://codepen.io/alexsuch/pen/jhfdB

```
});
```

```
<div ng-app="httpModule" ng-controller="MainCtrl">
  <h1>Resultado</h1>
  <hr />
</div>
```

EJERCICIO

> Genera un pequeño programa, similar al del ejemplo, que contenga un botón y, al presionarlo, realice una petición `DELETE` para eliminar el POST con id=1.

JSONP

Realiza una petición `JSONP`.

Parámetros:

- `url`: URL destino. El nombre del *callback* debe ser, obligatoriamente, `JSON_CALLBACK`
- `config`: objeto de configuración opcional.

En este ejemplo[72] haremos uso del servicio `$http.jsonp` para obtener los datos de un post.

```
angular
  .module('httpModule', [])
  .controller('MainCtrl', function($scope, $http){
    $http.jsonp( 'http://jsonplaceholder.typicode.com/posts/1?
callback=JSON_CALLBACK')
    .success(function(data){
      $scope.data = data;
    })
    .error(function(data){
      alert('Se ha producido un error')
    });
});
```

```
<div ng-app="httpModule" ng-controller="MainCtrl">
  <h1>Resultado</h1>
  <pre>
{{ data | json }}
  </pre>
  <hr />
</div>
```

EJERCICIO

> Genera un pequeño programa, similar al del ejemplo, que contenga un botón y, al presionarlo, realice una petición `JSONP` para obtener los datos del usuario con id=1.

[72] http://codepen.io/alexsuch/pen/cyavD

11.2. Integración con servicios RESTful: el servicio $resource

Como hemos visto en los ejemplos anteriores, el uso habitual de los servicios RESTful es para exponer operaciones CRUD, haciéndolas accesibles a través de una URL que acepta diferentes métodos HTTP.

El servicio $http nos da la posibilidad de interactuar con este tipo de servicios de manera sencilla. Sin embargo, disponemos de otro servicio, `$resource`, que nos permite hacer lo mismo eliminando además el código redundante.

El servicio `$resource` se distribuye en un módulo separado del *core* de AngularJS llamado `ngResource`. Es por ello que tendremos que descargarnos su código fuente y declarar una dependencia con este módulo donde lo vayamos a utilizar.

Para probarlo, seguiremos haciendo uso de los servicios de ejemplo de `jsonplaceholder`.

En primer lugar, crearemos un `resource` para la colección de usuarios del servicio:

```
 var User = $resource('http://jsonplaceholder.typicode.com/users/:id',
{id:'@id'});
```

A partir de esta URL, el servicio `$resource` creará par nosotros una serie de métodos para interactuar con el servicio RESTful.

Si nos centramos en la sintaxis de la declaración, vemos que recibe dos parámetros:

El primero es obligatorio, y consiste en una URL que puede estar parametrizada. Los parámetros irán siempre prefijados por el símbolo de los dos puntos `:`, de igual manera que hacíamos con los servicios de routing.

En cuanto al segundo parámetro, es opcional y consiste en el conjunto de valores por defecto para los parámetros de la URL. Podemos sobreescribirlos luego en las llamadas a métodos concretos.

Si alguno de los parámetros es una función, se ejecutará siempre antes de cada uso.

En caso de que en la URL parametrizada no tenga alguno de los parámetros, se pasará como parametro de búsqueda en la URL. Ejemplo: para la URL `/camera/:brand` y los parámetros `{brand:'canon', filter:'EOS 1100d'}`, obrendríamos la URL `/camera/canon?filter=EOS%201100D`

Si el valor del parámetro va precedido por una arroba `@`, entonces el valor de ese parámetro se extraerá del objeto que pasemos cuando invoquemos una acción, como veremos más adelante en los ejemplos.

El servicio acepta también un tercer parámetro, que veremos tras los ejemplos.

Volviendo al código que hemos generado

```
 var User = $resource('http://jsonplaceholder.typicode.com/users/:id',
{id:'@id'});
```

Veamos las operaciones que podemos realizar con él

Query

Forma: `User.query(params, successCallback, errorCallback)`

Realiza una petición `GET`, y espera recibir un array de ítems en la respuesta JSON.

Como vemos en el siguiente ejemplo[73], el atributo `params` es opcional, así como el la función de callback de error:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id:'@id'}
  );

  var userList = User.query(function(userList) {
    $scope.userList = userList;
  });
});
```

```
<div ng-app="restful" ng-controller="MainCtrl">
  <h3>Query</h3>
  <pre>{{ userList | json}}</pre>
</div>
```

Aunque, si queremos, podemos acceder a la promesa de resultados que genera la petición del servicio `$http` de la siguiente manera[74]:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id:'@id'}
  );

  var userList = User
    .query().$promise
    .then(function(userList) {
      $scope.userList = userList;
    });
});
```

Get

Forma: `User.get(params, successCallback, errorCallback)`

[73] http://codepen.io/alexsuch/pen/wystx
[74] http://codepen.io/alexsuch/pen/oHCKD

Realiza una petición GET al servidor, y espera recibir un objeto como resultado de la respuesta JSON.

Ejemplo[75]:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id:'@id'}
  );

  var user = User.get({id:1}, function(user) {
    $scope.user = user;
  });
});
```

```
<div ng-app="restful" ng-controller="MainCtrl">
  <h3>GET</h3>
  <pre>{{ user | json}}</pre>
</div>
```

En este caso, hemos pasado un objeto como primer parámetro que tiene el atributo `id`. Éste reemplazará el valor en el *template* de la url por el valor `1`.

Save

Forma: `User.save(params, payloadData, successCallback, errorCallback)`.

Envía una petición POST al servido. El cuerpo de la petición será el objeto que pasemos en el atributo `payloadData`.

Ejemplo[76]:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var userToSave = {
    "id": 1,
    "name": "Winchester McFly",
    "username": "wmf",
    "email": "wmf@hdh.com",
  };

  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id:'@id'}
  );

  User.save(
    userToSave,
```

[75] http://codepen.io/alexsuch/pen/lCsmA
[76] http://codepen.io/alexsuch/pen/pEcjw

```
    function(){
      $scope.message = 'usuario guardado con éxito';
    },
    function(){
      $scope.message = 'error al guardar';
    }
  );
});
```

```
<div ng-app="restful" ng-controller="MainCtrl">
  <h3>Save</h3>
  <pre>{{ message | json}}</pre>
</div>
```

En este caso, hemos introducido función de *callback* de error, ya que la API no nos permite realizar peticiones POST.

Delete

Formas:

- `User.delete(params, successCallback, errorCallback)`
- `User.remove(params, successCallback, errorCallback)`

Realiza una petición HTTP DELETE al servidor. Ejemplo[77]:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id:'@id'}
  );

  User.delete(
    {id:1},
    function(){
      $scope.message = 'Usuario eliminado correctamente'
    },
    function(){
      $scope.message = 'Eror al eliminar'
    }
  );
});
```

```
<div ng-app="restful" ng-controller="MainCtrl">
  <h3>Save</h3>
  <pre>{{ message | json}}</pre>
</div>
```

[77] http://codepen.io/alexsuch/pen/wAGpq

Definiendo acciones nuevas

Los métodos vistos (`query`, `get`, `save` y `delete`) son los únicos métodos que proporciona el servicio $resource, con el que podríamos comunicarnos con una gran cantidad de servicios RESTful.

Pero, ¿qué pasa si me comunico con una API que usa POST para guardar ítems nuevos, mientras espera PUT para actualizar ítems existentes? ¿Ya no es válido el servicio `$resource`?

Aunque no viene un método PUT por defecto en el servicio, sí que tenemos la posibilidad de crearlo. Es aquí donde entra en juego el tercer parámetro que habíamos obviado hasta ahora en la creación del servicio.

En él podemos definir nuevas acciones en nuestro servicio. Se trata de un *hash* donde declararemos todas las acciones *custom* que queramos añadir. La documentación de AngularJS[78] detalla al completo todos los parámetros que recibe. Nosotros, declararemos una función `update`[79] que realizará una petición PUT al servidor:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var userToUpdate = {
    "id": 1,
    "name": "Winchester McFly",
    "username": "wmf",
    "email": "wmf@hdh.com",
  };

  var User = $resource(
    'http://jsonplaceholder.typicode.com/users/:id',
    {id:'@id'},
    {
      update: {method:'PUT'}
    }
  );

  User.update(
    userToUpdate,
    function(data){
      $scope.message = data;
    },
    function(){
      $scope.message = 'error al actualizar';
    }
  );
});
```

```
<div ng-app="restful" ng-controller="MainCtrl">
  <h3>Save</h3>
  <pre>{{ message | json}}</pre>
</div>
```

[78] https://docs.angularjs.org/api/ngResource/service/$resource
[79] http://codepen.io/alexsuch/pen/sgJll

EJERCICIO

Adapta los ejemplos para conseguir aplicaciones que hagan lo mismo con comentarios (`query`, `get`, `update`).

Métodos a nivel de instancia

Puede que haya llamado la atención la declaración `var User = $resource(…)`, por haber usado mayúsculas. Esto es porque `$resource` genera una clase, y todos los métodos que hemos visto los hemos invocado a nivel de constructor.

Sin embargo, también podemos crear instancias de la clase `User`, lo que expone métodos a nivel de dicha instancia. Los métodos serán los mismos, pero prefijados por el símbolo del dólar `$`.

Ejemplo[80]:

```
angular.module('restful', ['ngResource'])
.controller('MainCtrl', function($scope, $resource){
  var data = {
    "id": 1,
    "name": "Winchester McFly",
    "username": "wmf",
    "email": "wmf@hdh.com",
  };

  var User = $resource('http://jsonplaceholder.typicode.com/users/:id',
 {id:'@id'}, {update: {method:'PUT'}});

  var u1 = new User(data);
  var u2 = new User(data);
  var u3 = new User(data);

  u1.$delete(
    function(res){ $scope.message1 = res; },
    function(res){ $scope.message1 = 'error al eliminar'; }
  );

  u1.$save(
    function(res){ $scope.message2 = res; },
    function(res){ $scope.message2 = 'error al guardar'; }
  );

  u1.$update(
    function(res){ $scope.message3 = res; },
    function(res){ $scope.message3 = 'error al actualizar'; }
  );
});
```

```
<div ng-app="restful" ng-controller="MainCtrl">
  <h3>Delete</h3>
  <pre>{{ message1 | json}}</pre>

  <h3>Save</h3>
```

[80] http://codepen.io/alexsuch/pen/icwyd

```
  <pre>{{ message2 | json}}</pre>

  <h3>Update</h3>
  <pre>{{ message3 | json}}</pre>
</div>
```

11.3. Interceptores

El servicio `$http` de AngularJS nos permite registrar interceptores que se ejecutarán en cada petición. Éstos resultan muy útiles cuando queremos realizar algún tipo de procesamiento sobre todas, o prácticamente todas las peticiones.

Supongamos que queremos comprobar cuándo tenemos permisos para realizar una petición. Para ello, podemos definir un interceptor que comprueba el código de estado de la respuesta y, si es un 401 (*HTTP 401 Unauthorized*), relanza lanza un evento indicando que se está realizando una operación no autorizada. Además, modificará todas las peticiones que enviemos, añadiendo las cabeceras de autorización básica.

```
angular
  .module('auth', [])
  .factory('AuthService', ['$log', function ($log) {
    var instance = {};
    var authServiceLastDate = new Date();
    var userData = null;
    var authToken = null;

    var doCheck = function(){
      if((new Date()).add(-30).minutes().getTime() > authServiceLastDate)
 {
        $log.debug('Session expired.');
        authServiceLastDate = null;
        userData = null;
      }

      authServiceLastDate = (new Date()).getTime();
    };

    instance.setUserData = function (userData) {
      authServiceLastDate = (new Date()).getTime();
      userData = userData;
    };

    instance.getUserData = function () {
      doCheck();
      return userData;
    };

    instance.deleteUserData = function () {
      userData = null;
    };

    instance.createBasicAuthToken = function(login, password) {
      return btoa(login + ':' + password);
    };
```

```
    instance.setToken = function (token) {
      authServiceLastDate = (new Date()).getTime();
      authToken = token;
    };

    instance.getToken = function () {
      doCheck();
      return authToken;
    };

    instance.deleteToken = function () {
      authToken = null;
    };

    return instance;
  }]);
  .factory('AuthInterceptor', ['$rootScope', '$q', 'AuthService', function
($rootScope, $q, AuthService) {
    var instance = {};

    instance.request = function(config) {
      config.headers = config.headers || {};

      if (!!AuthService.getToken()) {
        config.headers.Authorization = 'Basic ' + AuthService.getToken();
      } else {
        delete config.headers.Authorization;
      }

      return config;
    };

    instance.response = function(response) {
      if (response.status === 401) {
        AuthService.deleteUserData();
        AuthService.deleteToken();
        $rootScope.$emit('auth.unauthorized', []);
      }

      if(response.data.status && response.data.status === 'ERROR') { //
Force error
        return $q.reject(response);
      }

      return response;
    };

    return instance;
  }])
  .config(function($httpProvider){
    $httpProvider.interceptors.push('AuthInterceptor');
  });
```

Los interceptores son servicios de tipo factoría que registramos en el `$httpProvider`, añadiéndolos a la cola `$httpProvider.interceptors`. Al hacerse en un `provider`, tenemos que realizar esta operación en la fase de configuración.

Hay dos tipos de interceptores, y dos tipos de interceptores de rechazo:

- `request`: estos interceptores reciben como parámetro un objeto http `config`. Podemos modificar este objeto `config`, o bien crear uno nuevo. Se espera que esta función devuelva un objeto `config` (bien sea el existente o el nuevo) o una promesa que contenga el objeto `config`.
- `requestError`: este interceptor se llama cuando un interceptor previo lanza un error o se resuelve con un rechazo.
- `response`: estos interceptores reciben como parámetro un objeto http `response`. Podemos modificar este objeto `response` o crear uno nuevo. Se espera que esta función devuelva un objeto `response` (bien sea el existente o el nuevo) o una promesa que contenga el objeto `response`.
- `responseError`: este interceptor se llama cuando un interceptor previo lanza un error o se resuelve con un rechazo.

12. Automatización y testing

A lo largo de este capítulo iremos echando un vistazo a las herramientas de las que disponemos para poder seguir un *workflow* de desarrollo en AngularJS, que integre automatización y testing de nuestras aplicaciones.

Partimos de la base de que tenemos instalado en nuestro equipo tanto node.js[81] como npm[82].

12.1. Instalación de Grunt CLI

Grunt Command Line Interface (Grunt CLI) es un módulo de node.js que nos permite ejecutar tareas de Grunt en nuestro proyecto, vía línea de comandos. Así, podremos ejecutar cada tarea relacionada con el proceso de desarrollo de nuestra aplicación (verificación de sintaxis, ejecución de los tests unitarios, minificación, …). De esta manera, grunt se convierte en el único asistente que necesitamos para cubrir las necesidades de nuestro proyecto.

Podemos instalar el Grunt CLI a través de npm:

```
npm install -g grunt-cli
```

La opción *-g* hace que `grunt-cli` se instale de manera **global** y podremos ejecutarla a través del comando `grunt`. Grunt necesita una serie de componentes adicionales, que se instalarán de manera local a nuestro proyecto.

12.2. Instalación de Bower

Otro elemento global que necesitaremos es Bower[83]. Bower es a las librerías JavaScript de front-end lo que NPM a las librerías de backend de node.js. Este gestor de paquetes nos puede descargar librerías como AngularJS, ui-router, jQuery, etc. De manera que ya no es necesario irse al sitio web del framework/librería para descargarnos lo que necesitemos.

Bower se instala de manera similar a como hemos hecho para Grunt CLI:

```
npm install -g bower
```

12.3. Estructura inicial de nuestro proyecto

Gestión de dependencias

Vamos a definir las dependencias de nuestro proyecto. Necesitaremos una serie de librerías de backend, que gestionará npm, y de frontend, que gestionará Bower. Ambas herramientas necesitan un fichero de configuración JSON que define estas dependencias.

El fichero de configuración de node.js se llama `package.json`, y podemos inicializarlo lanzando, desde consola, el comando `npm init`:

```
$ npm init
```

81 http://nodejs.org/
82 https://www.npmjs.org/
83 http://bower.io/

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (angular-automation-testing)
version: (0.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Volumes/MSL64/tmp/angular-automation-testing/
package.json:

{
  "name": "angular-automation-testing",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this ok? (yes)
```

El fichero de configuración de bower se llama `bower.json`, y se inicializa con el comando `bower init`:

```
$ bower init
? name: angular-automation-testing
? version: 0.0.0
? description:
? main file:
? what types of modules does this package expose?:
? keywords:
? authors: Alejandro Such Berenguer <alejandro.such@gmail.com>
? license: MIT
? homepage:
? set currently installed components as dependencies?: Yes
? add commonly ignored files to ignore list?: Yes
? would you like to mark this package as private which prevents it from
 being accidentally published to the registry?: No

{
```

```
  name: 'angular-automation-testing',
  version: '0.0.0',
  authors: [
    'Alejandro Such Berenguer <alejandro.such@gmail.com>'
  ],
  license: 'MIT',
  ignore: [
    '**/.*',
    'node_modules',
    'bower_components',
    'test',
    'tests'
  ]
}

? Looks good?: Yes
```

Aunque vamos a querer introducir en nuestro sistema de control de versiones estos ficheros de configuración, vamos a querer ignorar las dependencias descargadas por bower y npm:

```
$ echo "node_modules/" >> .gitignore
$ echo "bower_components/" >> .gitignore
```

Ahora las dependencias. En la parte de front-end vamos a instalar las dependencias vistas en las sesiones de este módulo:

- AngularJS
- ui-router

```
$ bower install angular --save
$ bower install angular-ui-router --save
```

Añadimos la opción `-g`, para que las dependencias aparezcan en el fichero de configuración:

```
{
  "name": "angular-automation-testing",
  "version": "0.0.0",
  "authors": [
    "Alejandro Such Berenguer <alejandro.such@gmail.com>"
  ],
  "license": "MIT",
  "ignore": [
    "**/.*",
    "node_modules",
    "bower_components",
    "test",
    "tests"
  ],
  "dependencies": {
    "angular": "~1.3.2",
    "angular-ui-router": "~0.2.11"
  }
```

```
}
```

En cuanto a las dependencias de grunt, se instalan de la siguiente manera:

```
$ npm install angular-mocks --save-dev
$ npm install grunt --save-dev
$ npm install grunt-exec --save-dev
$ npm install grunt-contrib-clean --save-dev
$ npm install grunt-contrib-jshint --save-dev
$ npm install grunt-contrib-watch --save-dev
$ npm install grunt-contrib-concat --save-dev
$ npm install grunt-contrib-copy --save-dev
$ npm install grunt-contrib-uglify --save-dev
$ npm install karma --save-dev
$ npm install grunt-karma --save-dev
$ npm install karma-jasmine --save-dev
$ npm install load-grunt-tasks --save-dev
$ npm install karma-phantomjs-launcher --save-dev
$ npm install jquery --save-dev
```

También podemos ver que esas dependencias han aparecido en el fichero `package.json`:

```
{
  "name": "angular-automation-testing",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "angular-mocks": "^1.3.2",
    "grunt": "^0.4.5",
    "grunt-contrib-clean": "^0.6.0",
    "grunt-contrib-concat": "^0.5.0",
    "grunt-contrib-copy": "^0.7.0",
    "grunt-contrib-jshint": "^0.10.0",
    "grunt-contrib-uglify": "^0.6.0",
    "grunt-contrib-watch": "^0.6.1",
    "grunt-exec": "^0.4.6",
    "grunt-karma": "^0.9.0",
    "jquery": "^2.1.3",
    "karma": "^0.12.24",
    "karma-jasmine": "^0.2.3",
    "karma-phantomjs-launcher": "^0.1.4",
    "load-grunt-tasks": "^1.0.0"
  }
}
```

Utilizaremos la opción `--save-dev` para indicar que todas estas dependendencias son depenedencias de desarrollo, y no las utilizaremos nunca en un entorno de producción, ni son necesarias para que la aplicación se ejecute.

Muy importante la librería `angular-mocks`[84]. Ésta nos dará soporte para inyectar y *mockear* servicios de AngularJS en nuestros tests unitarios. También extiende varios servicios del *core* de AngularJS para que sean controlados de manera síncrona en nuestros tests (como veremos, por ejemplo, a la hora de hacer *mocks* de servicios HTTP).

Testing de filtros: nuestro primer test

Probaremos nuestra infraestructura. Para ello, seguiremos el paradigma *TDD*, diseñando un test para un filtro. El filtro se llamará `textOrDefault`, y devolverá la cadena que se le pase. En caso de no pasarse una cadena, se devolverá un valor por defecto (`-`), o el valor que se le pase como atributo (ej: `N/D`, `desconocido`, etc).

El fichero será `test/filters/textOrDefaultSpec.js`:

```
'use strict';

describe('filter: textOrDefault', function () {
  var textOrDefault;

  //Inicializar el módulo indicado antes de caa test
  beforeEach(module('filters.textordefault'));

  //Inyección de dependencias textOrDefault se apuntará al filtro
 inyectado
  beforeEach(inject(function (_textOrDefaultFilter_) {
    textOrDefault = _textOrDefaultFilter_;
  }));

  it("should return '-'", function () {
    expect(textOrDefault(null)).toBe('-');
    expect(textOrDefault('')).toBe('-');
    expect(textOrDefault('   ')).toBe('-');
  });

  it("should return 'N/D'", function () {
    expect(textOrDefault(null, 'N/D')).toBe('N/D');
    expect(textOrDefault('', 'N/D')).toBe('N/D');
    expect(textOrDefault('  \n\t ', 'N/D')).toBe('N/D');
  });

  it("should return the same value", function () {
    var hello = 'hello';
    expect(textOrDefault(hello, 'N/D')).toBe(hello);
    expect(textOrDefault(hello)).toBe(hello);

    var helloWithSpaces = '   hello   ';
    expect(textOrDefault(helloWithSpaces, 'N/D')).toBe(helloWithSpaces);
    expect(textOrDefault(helloWithSpaces)).toBe(helloWithSpaces);
  });
});
```

Como su nombre indica, la función `beforeEach` es llamada antes de que se ejecute cada test dentro del `describe`.

[84] https://docs.angularjs.org/api/ngMock

Utilizaremos esta función `beforeEach` para cargar el el módulo donde se encuentra nuestro filtro.

Definimos cada uno de nuestros tests dentro de una función `if`. Y ahí utilizaremos *expectations*, con la función `expect`. Ésta recibe un valor, llamado **valor real**, que se encadenará con una función de *matching* para compararlo con el valor esperado.

La página de introducción a Jasmine[85] dispone de ejemplos de todos y cada uno de los matchers por defecto.

Ahora, debemos establecer la configuración de Karma[86], para poder lanzar el test. Para ello, en la raíz del proyecto lanzaremos el comando `karma init`:

```
$ karma init

Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> jasmine

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next
 question.
> PhantomJS
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
> test/**/*Spec.js
> src/**/*.js

Should any of the files included by the previous patterns be excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>

Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> yes


Config file generated at "[RUTA_DEL_PROYECTO]/karma.conf.js".
```

Esta inicialización nos habrá creado el fichero `karma.conf.js`. Modificaremos el atributo `files` (listado de ficheros que se cargarán en el navegador en el momento de realizar los tests), dejándolo de la siguiente manera:

```
// list of files / patterns to load in the browser
```

[85] http://jasmine.github.io/edge/introduction.html
[86] http://karma-runner.github.io/0.12/index.html

```
files: [
  'bower_components/angular/angular.js',
  'node_modules/angular-mocks/angular-mocks.js',
  'bower_components/angular-ui-router/release/angular-ui-router.js',
  'src/**/*.js',
  'test/**/*Spec.js'
],
```

Si lanzamos los tests con `karma start karma.conf.js`, veremos cómo se levanta el navegador PhantomJS y nos devuelve un error. Esto se debe a que el test se ha lanzado, pero no se encuentra el módulo a testear. Crearemos el fichero `src/filters/testOrDefault.js`:

```
(function () {
    'use strict';
    angular.module('filters.textordefault', [])
        .filter('textOrDefault', function () {
            return function (input, defaultValue) {
                return input;
            };
        });
})();
```

Volviendo a lanzar los tests, veremos que ahora éstos fallan porque el filtro no está devolviendo los valores que esperábamos.

Modificaremos el código de nuestro filtro para que realice la funcionalidad esperada:

```
(function () {
    'use strict';
    angular.module('filters.textordefault', [])
        .filter('textOrDefault', function () {
            return function (input, defaultValue) {
                defaultValue = defaultValue || '-';

                if (!input)
                    return defaultValue;

                if (!angular.isString(input)) {
                    if (input.toString) {
                        input = input.toString();
                    } else {
                        return defaultValue;
                    }
                }

                if (input.trim().length > 0) {
                    return input;
                }

                return defaultValue;
            };
        });
})();
```

Al realizar estos cambios, los tests pasarán exitosamente.

Una cosa que podemos ver es que se detectan los cambios "en caliente". A medida que modificamos el código de nuestro filtro, si salvamos, se volverán a lanzar los tests.

Testing de servicios

Veamos ahora cómo testear un servicio de cualquier tipo (provider, factory o service). Haremos nuestro ejemplo con un provider, ya que tiene un componente de configuración que el resto de servicios no tiene.

La idea es crear un servicio de validaciones. Tendremos por una parte una serie de validaciones predefinidas, y además podremos añadir los validadores *custom* al servicio.

El código de nuestro test (`test/providers/expertoJeeValidationProviderSpec.js`) será el siguiente:

```
'use strict';

describe('provider: expertoJeeValidation', function () {
    var validationProvider;
    var undefinedVar;
    var validationService;

    beforeEach(module('providers.validation'));

    beforeEach(function () {
        // Creamos un módulo de pega, al que inyectamos el provider y
 definimos una función de configuración
        var fakeModule = angular
            .module('test.app.config', function(){}).config(function
 (expertoJeeValidationProvider) {
                validationProvider = expertoJeeValidationProvider;


 validationProvider.addConstraint('customConstraint', function (value,
 needsToBeFive) {
                    if (needsToBeFive) {
                        return value === 5;
                    }

                    return true;
                });
            });

        // Cargamos los módulos
        module('test.app.config');
   });

  beforeEach(
        //Inyectar los servicios en los tests
        inject(function(_expertoJeeValidation_){
          validationService = _expertoJeeValidation_;
        })
    );

    it('tests the providers has been injected', function () {
```

```
        expect(validationProvider).not.toBeUndefined();
        expect(validationService).not.toBeUndefined();
    });

    it('tests the blank constraint', function () {
        var blankConstraint = validationService.blank;

        expect(blankConstraint('', true)).toBe(true);
        expect(blankConstraint('', false)).toBe(false);

        expect(blankConstraint(undefinedVar, true)).toBe(true);
        expect(blankConstraint(undefinedVar, false)).toBe(false);

        expect(blankConstraint(null, true)).toBe(true);
        expect(blankConstraint(null, false)).toBe(false);

        expect(blankConstraint('hello', true)).toBe(true);
        expect(blankConstraint('hello', false)).toBe(true);
    });

    it('tets the creditCard constraint', function () {
        var creditCardConstraint = validationService.creditCard;
        var testFn = function () {
            creditCardConstraint(null, true);
        };

        expect(testFn).toThrow('CreditCard constraint: Not implemented
yet');
    });

    it('tests the email constraint', function () {
        var emailConstraint = validationService.email;

        expect(emailConstraint('', true)).toBe(false);
        expect(emailConstraint('', false)).toBe(true);

        expect(emailConstraint(undefinedVar, true)).toBe(false);
        expect(emailConstraint(undefinedVar, false)).toBe(true);

        expect(emailConstraint(null, true)).toBe(false);
        expect(emailConstraint(null, false)).toBe(true);


        expect(emailConstraint('admin', true)).toBe(false);
        expect(emailConstraint('admin', false)).toBe(true);

        expect(emailConstraint('admin@', true)).toBe(false);
        expect(emailConstraint('admin@', false)).toBe(true);

        expect(emailConstraint('admin@admin', true)).toBe(true);
        expect(emailConstraint('admin@admin', false)).toBe(true);

        expect(emailConstraint('admin@admin.', true)).toBe(false);
        expect(emailConstraint('admin@admin.', false)).toBe(true);

        expect(emailConstraint('admin@admin.com', true)).toBe(true);
        expect(emailConstraint('admin@admin.com', false)).toBe(true);
    });
```

```
    it('tests the inList constraint', function () {
        var inListConstraint = validationService.inList;

        var testFn = function () {
            inListConstraint(null, true);
        };

        var testFn2 = function () {
            inListConstraint(null, 1);
        };

        var testFn2 = function () {
            inListConstraint(null, 'hello');
        };

        var testFn3 = function () {
            inListConstraint(null, { name: 'John', lastName: 'Locke'});
        };

        expect(testFn).toThrow('InList constraint only applies to
 Arrays');
        expect(testFn2).toThrow('InList constraint only applies to
 Arrays');
        expect(testFn3).toThrow('InList constraint only applies to
 Arrays');

        expect(inListConstraint('a', ['a', 'b', 'c'])).toBe(true);
        expect(inListConstraint('d', ['a', 'b', 'c'])).toBe(false);
        expect(inListConstraint(1, ['a', 'b', 'c'])).toBe(false);
        expect(inListConstraint(1, ['1', '2', '3'])).toBe(false);
        expect(inListConstraint(1, [1, 2, 3])).toBe(true);
        expect(inListConstraint(undefinedVar, [1, 2, 3])).toBe(false);
        expect(inListConstraint(undefinedVar,
 ['a', 'b', 'c'])).toBe(false);
        expect(inListConstraint(null, ['a', 'b', 'c'])).toBe(false);
        expect(inListConstraint(null, ['a', 'b', 'c', null])).toBe(true);
    });

    it('tests the regex constraint', function () {
        var matchesConstraint = validationService.matches;
        var emailRegex = '^[a-z0-9!#$%&\'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-
z0-9-]*[a-z0-9])?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$';

        var testFn = function () {
            matchesConstraint(5, emailRegex);
        };

        var testFn2 = function () {
            matchesConstraint(undefinedVar, emailRegex);
        };

        var testFn3 = function () {
            matchesConstraint(null, emailRegex);
        };

        var testFn4 = function () {
```

```
            matchesConstraint([], emailRegex);
        };

        var testFn5 = function () {
            matchesConstraint({}, emailRegex);
        };

        expect(testFn).toThrow('Matches constraint only applies to
Strings');
        expect(testFn2).toThrow('Matches constraint only applies to
Strings');
        expect(testFn3).toThrow('Matches constraint only applies to
Strings');
        expect(testFn4).toThrow('Matches constraint only applies to
Strings');
        expect(testFn5).toThrow('Matches constraint only applies to
Strings');

        expect(matchesConstraint('', emailRegex)).toBe(false);
        expect(matchesConstraint('admin', emailRegex)).toBe(false);
        expect(matchesConstraint('admin@', emailRegex)).toBe(false);
        expect(matchesConstraint('admin@admin', emailRegex)).toBe(true);
        expect(matchesConstraint('admin@admin.', emailRegex)).toBe(false);
        expect(matchesConstraint('admin@admin.com',
emailRegex)).toBe(true);

    });

    it('tests the max constraint', function () {
        var maxConstraint = validationService.max;
        var throwErr = 'Max constraint only applies to numbers';

        var testFn = function () {
            maxConstraint('5', 4);
        };

        var testFn2 = function () {
            maxConstraint(undefinedVar, 4);
        };

        var testFn3 = function () {
            maxConstraint(null, 4);
        };

        var testFn4 = function () {
            maxConstraint([], 4);
        };

        var testFn5 = function () {
            maxConstraint({}, 4);
        };

        expect(testFn).toThrow(throwErr);
        expect(testFn2).toThrow(throwErr);
        expect(testFn3).toThrow(throwErr);
        expect(testFn4).toThrow(throwErr);
        expect(testFn5).toThrow(throwErr);
```

```
        expect(maxConstraint(1, 4)).toBe(true);
        expect(maxConstraint(4, 4)).toBe(true);
        expect(maxConstraint(5, 4)).toBe(false);
    });

    it('tests the maxSize constraint', function () {
        var maxSizeConstraint = validationService.maxSize;
        var throwErr = 'MaxSize constraint only applies to Arrays and
Strings';
        var throwErr2 = 'Argument maxSize should be a number';

        var testFn = function () {
            maxSizeConstraint(undefinedVar, '');
        };

        var testFn2 = function () {
            maxSizeConstraint(undefinedVar, 4);
        };

        var testFn3 = function () {
            maxSizeConstraint(null, 4);
        };

        var testFn5 = function () {
            maxSizeConstraint({}, 4);
        };

        expect(testFn).toThrow(throwErr2);
        expect(testFn2).toThrow(throwErr);
        expect(testFn3).toThrow(throwErr);
        expect(testFn5).toThrow(throwErr);

        expect(maxSizeConstraint('hello', 4)).toBe(false);
        expect(maxSizeConstraint('hello', 5)).toBe(true);
        expect(maxSizeConstraint('hello', 6)).toBe(true);

        expect(maxSizeConstraint([1, 2, 3, 4, 5], 4)).toBe(false);
        expect(maxSizeConstraint([1, 2, 3, 4, 5], 5)).toBe(true);
        expect(maxSizeConstraint([1, 2, 3, 4, 5], 6)).toBe(true);
    });

    it('tests the min constraint', function () {
        var minConstraint = validationService.min;
        var throwErr = 'Min constraint only applies to numbers';

        var testFn = function () {
            minConstraint('5', 4);
        };

        var testFn2 = function () {
            minConstraint(undefinedVar, 4);
        };

        var testFn3 = function () {
            minConstraint(null, 4);
        };
```

```
        var testFn4 = function () {
            minConstraint([], 4);
        };

        var testFn5 = function () {
            minConstraint({}, 4);
        };

        expect(testFn).toThrow(throwErr);
        expect(testFn2).toThrow(throwErr);
        expect(testFn3).toThrow(throwErr);
        expect(testFn4).toThrow(throwErr);
        expect(testFn5).toThrow(throwErr);

        expect(minConstraint(1, 4)).toBe(false);
        expect(minConstraint(4, 4)).toBe(true);
        expect(minConstraint(5, 4)).toBe(true);
    });

    it('tests the minSize constraint', function () {
        var minSizeConstraint = validationService.minSize;
        var throwErr = 'MinSize constraint only applies to Arrays and
Strings';
        var throwErr2 = 'Argument minSize should be a number';

        var testFn = function () {
            minSizeConstraint(undefinedVar, '');
        };

        var testFn2 = function () {
            minSizeConstraint(undefinedVar, 4);
        };

        var testFn3 = function () {
            minSizeConstraint(null, 4);
        };

        var testFn5 = function () {
            minSizeConstraint({}, 4);
        };

        expect(testFn).toThrow(throwErr2);
        expect(testFn2).toThrow(throwErr);
        expect(testFn3).toThrow(throwErr);
        expect(testFn5).toThrow(throwErr);

        expect(minSizeConstraint('hello', 4)).toBe(true);
        expect(minSizeConstraint('hello', 5)).toBe(true);
        expect(minSizeConstraint('hello', 6)).toBe(false);

        expect(minSizeConstraint([1, 2, 3, 4, 5], 4)).toBe(true);
        expect(minSizeConstraint([1, 2, 3, 4, 5], 5)).toBe(true);
        expect(minSizeConstraint([1, 2, 3, 4, 5], 6)).toBe(false);
    });

    it('tests the notEqual constraint', function () {
        var notEqualConstraint = validationService.notEqual;
```

```
        var testFn = function () {
            notEqualConstraint(1, 1);
        };

        expect(testFn).toThrow('NotEqual constraint: Not implemented
yet');
    });

    it('tests the nullable constraint', function () {
        var nullableConstraint = validationService.nullable;

        expect(nullableConstraint('', true)).toBe(true);
        expect(nullableConstraint('', false)).toBe(true);

        expect(nullableConstraint(null, true)).toBe(true);
        expect(nullableConstraint(null, false)).toBe(false);

        expect(nullableConstraint(undefinedVar, true)).toBe(true);
        expect(nullableConstraint(undefinedVar, false)).toBe(false);
    });

    it('tests the numeric constraint', function () {
        var numericConstraint = validationService.numeric;
        var throwErr = 'Numeric constraint expects two arguments';

        var testFn = function () {
            numericConstraint('a');
        };

        expect(testFn).toThrow(throwErr);

        expect(numericConstraint(5, true)).toBe(true);
        expect(numericConstraint(5, false)).toBe(true);

        expect(numericConstraint(null, true)).toBe(false);
        expect(numericConstraint(null, false)).toBe(true);

        expect(numericConstraint(undefinedVar, true)).toBe(false);
        expect(numericConstraint(undefinedVar, false)).toBe(true);

        expect(numericConstraint('5', true)).toBe(false);
        expect(numericConstraint('5', false)).toBe(true);

        expect(numericConstraint([], true)).toBe(false);
        expect(numericConstraint([], false)).toBe(true);

        expect(numericConstraint({}, true)).toBe(false);
        expect(numericConstraint({}, false)).toBe(true);
    });

    it('tests the range constraint', function () {
        var rangeConstraint = validationService.range;
        var throwErr = 'Range constraint expects three arguments';
        var throwErr2 = 'All three values must be numbers';

        var testFn = function () {
```

```
            rangeConstraint('a', '1');
        };

        var testFn2 = function () {
            rangeConstraint('a', '1', 10);
        };

        expect(testFn).toThrow(throwErr);
        expect(testFn2).toThrow(throwErr2);

        expect(rangeConstraint(5, 0, 10)).toBe(true);
        expect(rangeConstraint(0, 0, 10)).toBe(true);
        expect(rangeConstraint(10, 0, 10)).toBe(true);
        expect(rangeConstraint(-1, 0, 10)).toBe(false);
        expect(rangeConstraint(11, 0, 10)).toBe(false);
    });

    it('tests the size constraint', function () {
        var sizeConstraint = validationService.size;
        var throwErr = 'Size constraint expects three arguments';
        var throwErr2 = 'Size constraint only applies to Arrays and
Strings';
        var throwErr3 = 'Start and end values must be numbers';

        var testFn = function () {
            sizeConstraint('a', '1');
        };

        var testFn2 = function () {
            sizeConstraint({}, 1, 10);
        };

        var testFn3 = function () {
            sizeConstraint('a', '1', 10);
        };

        expect(testFn).toThrow(throwErr);
        expect(testFn2).toThrow(throwErr2);
        expect(testFn3).toThrow(throwErr3);

        expect(sizeConstraint("hello", 0, 5)).toBe(true);
        expect(sizeConstraint("", 0, 5)).toBe(true);
        expect(sizeConstraint("hi", 0, 5)).toBe(true);
        expect(sizeConstraint("hello world", 0, 5)).toBe(false);

        expect(sizeConstraint([1, 2, 3, 4, 5], 0, 5)).toBe(true);
        expect(sizeConstraint([], 0, 5)).toBe(true);
        expect(sizeConstraint([1, 2], 0, 5)).toBe(true);

expect(sizeConstraint([1, 2, 3, 4, 5, 6, 7, 8, 9], 0, 5)).toBe(false);
    });

    it('tests the unique constraint', function () {
        var uniqueConstraint = validationService.unique;

        var throwErr = 'Unique constraint: not implemented yet';
```

```
        var testFn = function () {
            uniqueConstraint('a', true);
        };

        expect(testFn).toThrow(throwErr);
    });

    it('tests the url constraint', function () {
        var urlConstraint = validationService.url;

        var throwErr = 'Url constraint: expected 2 arguments';
        var throwErr2 = 'Url constraint: value expected to be a string';
        var throwErr3 = 'Url constraint: url expected to be a boolean';

        var testFn = function () {
            urlConstraint(1);
        };

        var testFn2 = function () {
            urlConstraint(1, true);
        };

        var testFn3 = function () {
            urlConstraint('1', 'true');
        };

        expect(testFn).toThrow(throwErr);
        expect(testFn2).toThrow(throwErr2);
        expect(testFn3).toThrow(throwErr3);

        expect(urlConstraint('www.ua.es', false)).toBe(true);
        expect(urlConstraint('asdf', true)).toBe(false);
        expect(urlConstraint('www.ua.es', true)).toBe(false);
        expect(urlConstraint('http://www.ua.es', true)).toBe(true);
    });

    it('tests a custom constraint', function () {
        var customConstraint = validationService.customConstraint;

        expect(customConstraint(5, true)).toBe(true);
        expect(customConstraint(4, false)).toBe(true);
        expect(customConstraint(54, true)).toBe(false);
    });

    it('should fail trying to override an existing constraint', function
() {
        var throwErr = 'Cannot override a default constraint';
        var testFn = function () {
            validationProvider.addConstraint('url', function (value,
needsToBeFive) {
                if (needsToBeFive) {
                    return value === 5;
                }

                return true;
            });
        };
```

```
        expect(testFn).toThrow(throwErr);
    });
});
```

La función `module` se utiliza para indicar al test que deberían prepararse los servicios del módulo indicado. El rol de este método es similar al de la directiva `ng-app` en una vista.

La función `inject`[87] tiene la responsabilidad de inyectar los servicios en nuestros tests.

Por su parte, el código del provider será:

```
(function () {
    'use strict';

    angular
        .module('providers.validation')
        .provider('expertoJeeValidation', function () {
            var instance = {};
            /**
             * Validates that a String value is not blank
             * @param value
             * @param blank
             * @returns {boolean}
             */
            instance.blank = function (value, blank) {
                if (typeof value !== 'undefined' && value !== null
 && typeof value !== 'string' && !(value instanceof String)) {
                    throw 'Blank constraint only applies to strings';
                }

                var isBlank = typeof value === 'undefined' || value ===
 null || value.length === 0 || !value.trim();

                if (!blank) {
                    return !isBlank;
                }

                return true;
            };

            /**
             * Validates that a String value is a valid credit card number
             * @param value
             * @param creditCard
             * @returns {boolean}
             */
            instance.creditCard = function (value, creditCard) {
                throw 'CreditCard constraint: Not implemented yet';

//                return false;
            };

            /**
```

[87] https://docs.angularjs.org/api/ngMock/function/angular.mock.inject

```
             * Validates that a String value is a valid email address.
             * @param value
             * @param email
             * @returns {boolean}
             */
            instance.email = function (value, email) {
                var emailRegex = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]
([a-z0-9-]*[a-z0-9])?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;

                if (email) {
                    return emailRegex.test(value);
                }

                return true;
            };

            /**
             * Validates that a value is within a range or collection of
 constrained values.
             * @param value
             * @param array
             * @returns {boolean}
             */
            instance.inList = function (value, array) {
                if (!(array instanceof Array)) {
                    throw 'InList constraint only applies to Arrays';
                }

                return array.indexOf(value) !== -1;
            };

            /**
             * Validates that a String value matches a given regular
 expression.
             * @param value
             * @param expr
             * @returns {boolean}
             */
            instance.matches = function (value, expr) {
                if (typeof value !== 'string' && !
(value instanceof String)) {
                    throw 'Matches constraint only applies to Strings';
                }

                var regexp = new RegExp(expr);

                return regexp.test(value);
            };

            /**
             * Validates that a value does not exceed the given maximum
 value.
             * @param value
             * @param max
             * @returns {boolean}
             */
```

```
            instance.max = function (value, max) {
                if (typeof value !== 'number' && !
(value instanceof Number)) {
                    throw 'Max constraint only applies to numbers';
                }

                return value <= max;
            };

            /**
             * Validates that a value's size does not exceed the given
 maximum value.
             * @param value
             * @param maxSize
             * @returns {boolean}
             */
            instance.maxSize = function (value, maxSize) {
                if (value instanceof Array || value instanceof String
 || typeof value === 'string') {
                    return value.length <= maxSize;
                }

                if (typeof maxSize !== 'number' && !
(maxSize instanceof Number)) {
                    throw 'Argument maxSize should be a number';
                }

                throw 'MaxSize constraint only applies to Arrays and
 Strings';
            };

            /**
             * Validates that a value does not fall below the given
 minimum value.
             * @param value
             * @param min
             * @returns {boolean}
             */
            instance.min = function (value, min) {
                if (typeof value !== 'number' && !
(value instanceof Number)) {
                    throw 'Min constraint only applies to numbers';
                }

                return value >= min;
            };

            /**
             * Validates that a value's size does not fall below the given
 minimum value.
             * @param value
             * @param minSize
             * @returns {boolean}
             */
            instance.minSize = function (value, minSize) {
```

```
                if (value instanceof Array || value instanceof String
 || typeof value === 'string') {
                    return value.length >= minSize;
                }

                if (typeof minSize !== 'number' && !
(minSize instanceof Number)) {
                    throw 'Argument minSize should be a number';
                }

                throw 'MinSize constraint only applies to Arrays and
 Strings';
            };

            /**
             * Validates that that a property is not equal to the
 specified value
             * @param value
             * @param otherValue
             * @returns {boolean}
             */
            instance.notEqual = function (value, otherValue) {
                throw 'NotEqual constraint: Not implemented yet';

//                return value !== otherValue;
            };

            /**
             * Allows a property to be set to null - defaults to true.
 Undefined is considered null in this constraint
             * @param value
             * @param nullable
             * @returns {boolean}
             */
            instance.nullable = function (value, nullable) {
                if (arguments.length !== 2) {
                    throw 'Constraint error. Must provide a boolean value
 for nullable';
                }

                if (!nullable) {
                    return value !== null && typeof value !== 'undefined';
                }

                return true;
            };

            /**
             * Ensures that the given value should be numeric
             * @param value
             * @param numeric
             */
            instance.numeric = function (value, numeric) {
                if (arguments.length !== 2) {
                    throw 'Numeric constraint expects two arguments';
                }
```

```
                var isNumeric = typeof value === 'number' ||
 value instanceof Number;

                if (numeric) {
                    return isNumeric;
                }

                return true;
            };

            /**
             * Ensures that a property's value occurs within a specified
 range
             * @param value
             * @param start
             * @param end
             * @returns {boolean}
             */
            instance.range = function (value, start, end) {
                if (arguments.length !== 3) {
                    throw 'Range constraint expects three arguments';
                }

                if (!instance.numeric(value, true) || !
instance.numeric(start, true) || !instance.numeric(end, true)) {
                    throw 'All three values must be numbers';
                }

                return value >= Math.min(start, end) && value
 <= Math.max(start, end);
            };

            /**
             * Restricts the size of a collection or the length of a
 String.
             * @param value
             * @param start
             * @param end
             * @returns {boolean}
             */
            instance.size = function (value, start, end) {
                if (arguments.length !== 3) {
                    throw 'Size constraint expects three arguments';
                }

                if (!instance.numeric(start, true) || !
instance.numeric(end, true)) {
                    throw 'Start and end values must be numbers';
                }

                if (value instanceof Array || value instanceof String
 || typeof value === 'string') {
                    return value.length >= Math.min(start, end) &&
 value.length <= Math.max(start, end);
                }
```

```
                throw 'Size constraint only applies to Arrays and
 Strings';
            };

            /**
             * Constrains a property as unique at the database level
             * @param value
             * @param unique
             * @returns {boolean}
             */
            instance.unique = function (value, unique) {
                throw 'Unique constraint: not implemented yet';

//                return false;
            };

            /**
             * Validates that a String value is a valid URL.
             * @param value
             * @param url
             * @returns {boolean}
             */
            instance.url = function (value, url) {
                if(arguments.length !== 2) {
                    throw 'Url constraint: expected 2 arguments';
                }

                if(typeof value !== 'string' && !
(value instanceof String)) {
                    throw 'Url constraint: value expected to be a string';
                }

                if(typeof url !== 'boolean' && !(url instanceof Boolean))
 {
                    throw 'Url constraint: url expected to be a boolean';
                }

                var urlRegex = /^(ftp|http|https):\/\/(\w+:{0,1}\w*@)?(\S
+)(:[0-9]+)?(\/|\/([\w#!:.?+=&%@!\-\/]))?$/;

                if (url) {
                    return urlRegex.test(value);
                }

                return true;
            };

            var defaultConstraints = [];
            for (var i in instance) {
                defaultConstraints.push(i);
            }

            this.addConstraint = function (constraintName, fn) {
                if (defaultConstraints.indexOf(constraintName) !== -1) {
                    throw 'Cannot override a default constraint';
                }
```

```
                instance[constraintName] = fn;
            };

            this.setErrorMessage = function (constraintName, message) {
                instance[constraintName + 'Message'] = message;
            };

            instance.getErrorMessage = function (constraintName) {
                return instance[constraintName + 'Message'];
            };

            this.getInstance = this.$get = function () {
                return instance;
            };
        });
})();
```

Lanzando ahora el test nos dará error. Esto se debe a que el módulo del provider no está correctamente definido. Lo corregiremos para que todo funcione correctamente:

```
angular
    .module('providers.validation', [])
```

Sobre el método `inject`

`inject` permite que el servicio a inyectar tenga su nombre habitual (ej: `$http`), o bien su nombre habitual, envuelto por guiones bajos (ej: `_$http_`). Estos guiones son ignorados por el inyector a la hora de resolver el nombre del servicio, y puede ser de gran utilidad si preferimos usar su nombre habitual en nuestros tests.

Así estos dos tests serían equivalentes, solo que en un caso mantendríamos el nombre del servicio en lugar de una variable con otro nombre:

```
describe('provider: expertoJeeValidation', function () {
    var expertoJeeValidation;

    beforeEach(module('providers.validation'));

    beforeEach(
        //Inyectar los servicios en los tests
        inject(function(_expertoJeeValidation_){
          expertoJeeValidation = _expertoJeeValidation_;
        })
    );

    // Resto del test. Usaremos 'expertoJeeValidation',
    // que es lo mismo que usaríamos en el código
    // de nuestra aplicación
}
```

```
describe('provider: expertoJeeValidation', function () {
    var theService;
```

```
    beforeEach(module('providers.validation'));

    beforeEach(
        //Inyectar los servicios en los tests
        inject(function(expertoJeeValidation){
          theService = expertoJeeValidation;
        })
    );

    //Resto del test. Usaremos 'theService'
}
```

Testing de controladores y *Mocking* de peticiones HTTP.

Vamos ahora a ver qué sería necesario para testear un controlador. Supondremos un controlador que expondrá en el `scope` un método llamado `getUsers`, que se conectará a un servicio HTTP (`/users`) y devolverá un listado de usuarios. El controlador también deberá contemplar posibles errores en la llamada al servicio.

Si se produjera algún error, existe una variable en el `scope` llamada `hasError` que pasaría a tener un valor cierto. Los usuarios se guardarán en una variable del `scope` llamada `users`.

El código del test (`test/controller/usersCtrlSpec.js`) será el siguiente:

```
'use strict';

describe('Controller: usersCtrl', function() {
    var scope, controller, httpBackend;

    // Inicializar el módulo antes de cada test
    beforeEach(module('expertojee.controllers'));

    // Inyección de dependencias, mockearemos $http con el servicio
 $httpBackend
    beforeEach(inject(function($rootScope, $controller, $httpBackend) {
        scope = $rootScope.$new();
        controller = $controller;
        httpBackend = $httpBackend;
    }));

    iit('should query the webservice', function() {
        // Definimos qué petición HTTP esperamos, y qué resultado queremos
 devolver
        httpBackend
          .expectGET('/users')
          .respond('[{"firstName": "Alejandro", "lastName": "Such"},
{"firstName": "Domingo", "lastName": "Gallardo"}]');

        // Inicializamos el controlador
        controller('usersCtrl', {'$scope': scope });

        //Llamamos al método del controlador
        scope.getUsers()

        // Responder a todas las peticiones HTTP
```

```
        httpBackend.flush();

        // Lanzamos scope.$apply() para que se resuelvan todas las
  promesas
        scope.$apply();

        // Evaluar los valores esperados
        expect(scope.users.length).toBe(2);
        expect(scope.hasError).toBe(false);
    });

    iit('should catch an error', function() {
        // Cuando se realice una petición a /users, responder con un error
  500
        httpBackend
          .expectGET('/users')
          .respond(500, null);

        // Inicializar el controlador
        controller('usersCtrl', {'$scope': scope });

        // Llamamos al método del controlador
        scope.getUsers()

        // Responder a todas las peticiones HTTP
        httpBackend.flush();

        // Lanzamos scope.$apply() para que se resuelvan todas las
  promesas
        scope.$apply();

        // Evaluar los valores esperados
        expect(scope.hasError).toBe(true);
        expect(scope.users).toBeNull();
    });
});
```

A destacar que cada test se define con la función `iit` en lugar de `it`. Si en algún momento introducimos alguna función `iit` el resto de funciones definidas con `it` serán ignoradas. Esto es cómodo si nos queremos centrar en algún test en concreto.

Vemos cómo inicializamos el controlador con el servicio `$controller`, e inyectándole un `scope` que hemos creado en la función `beforeEach` (`scope = $rootScope.$new()`).

Lo más importante es el uso del servicio $httpBackend[88]. Éste nos permite implementar llamadas falsas a un servicio y simular los resultados que queramos obtener en cada test. Al inicio de nuestro test escribimos el resultado que queremos probar en cada caso, y una vez llamada a la función que hace uso del servicio, deberemos realizar una llamada al método `$httpBackend.flush()` para que todas las llamadas al servicio `$http` que se hayan hecho en el controlador reciban su respuesta.

Aunque en el código que implementaremos no es necesario, en algunas ocasiones, dado que las peticiones http trabajan con promesas de resultados, tendremos que llamar a `scope.$digest()` o `scope.$apply()` para que los resultados pasen al `scope`.

[88] https://docs.angularjs.org/api/ngMock/service/$httpBackend

Un código de controlador que pasaría los dos tests escritos es (`src/controller/usersCtrl.js`):

```
(function(){
  'use strict';

  angular
      .module('expertojee.controllers', [])
      .controller('usersCtrl', function($scope, $http){
        $scope.hasError = false;
        $scope.users = null

        $scope.getUsers = function(){
          $http
            .get('/users')
            .success(function(users){
              $scope.users = users;
              $scope.hasError = false;
            })
            .catch(function(){
              $scope.users = null
              $scope.hasError = true;
            });
        };
      });
})();
```

Testing de directivas

Finalmente, veremos cómo podemos realizar tests unitarios de directivas. Aunque pueda parecer más difícil, veremos como el proceso es bastante similar a lo que hemos hecho hasta ahora.

El truco está en que necesitaremos *compilar* el código HTML. Para ello utilizaremos el servicio $compile[89]. Compilar consiste en introducir una cadena HTML en el ciclo de AngularJS, asociándole un *scope*.

Para testear una directiva vamos a tener que compilarla, realizar la tarea que tenemos que realizar (si fuese necesario), y finalmente invocar al método `$apply()` o `$digest()` del *scope* para que procesar los cambios.

Supongamos la siguiente directiva (`src/directive/scheduleEvent.js`):

```
(function () {
  'use strict';
  angular.module('directives.schedule', [])
  .directive('scheduleEvent', function () {
    return {
      restrict: 'E',
      scope: {
        event: '=',
        deleteAction: '&'
      },
```

[89] https://docs.angularjs.org/api/ng/service/$compile

```
      template: '<div="schedule-event"> ' +
      '<h2> ' +
      '<span ng-if="isToday"><i class="icon ion-ios7-time-outline"></i></
span> ' +
      '<span ng-if="!isToday"><i class="icon ion-ios7-calendar-outline"></
i> {{ event.date | date }} - </span> ' +
      '<span ng-bind="event.date | date:\'HH:mm\'"></span>. <span ng-
bind="event.title"></span> ' +
      '<p ng-if="showHolder" ng-bind="event.contact.name + \' \' +
 event.contact.middleName + \' \' + event.contact.lastName"></p> ' +
      '</h2> ' +
      '<a class="button" ng-click="deleteAction()">Eliminar</a> ' +
      '</div>',

      link: function (scope, element, attrs) {
        scope.isToday = scope.$eval(attrs.isToday);
        scope.showHolder = !!scope.event.contact && !scope.
$eval(attrs.hideContact);

        scope.$on('$destroy', function(){
          angular.element(element).remove();
        });
      }
    };
  });
})();
```

Ésta consiste en una entrada de agenda, que puede estar o no asociada a un contacto. Mostrará un botón "Eliminar" Acepta los siguientes atributos:

- *event*: Entrada de agenda. Objeto con los atributos *title* y *contact*. *contact* tiene, a su vez, los atributos *firstName*, *middleName* y *lastName*.
- *hideContact*: Ocultar el nombre del contacto.
- *isToday*: el evento es del día de hoy. Acepta los valores "true" o "false". _ *deleteAction*: acción a realizar cuando se hace click en el botón de eliminar

Como hemos comentado, en el test habrá que compilar primero un bloque HTML. Para ello, necesitaremos inyectar el servicio `$compile` antes de cada test. Como hemos dicho que este servicio asocia una cadena HTML a un *scope*, también haremos uso del `$rootScope`, donde definiremos la acción a realizar cuando hagamos click en el botón *delete*:

```
beforeEach(inject(function ($compile, $rootScope) {
  scope = $rootScope;
  compile = $compile;

  scope.deleteEvent = function () {
    console.log('deleting event');
  };
}));
```

En cada uno de nuestros tests, compilaremos el código HTML que deseemos probar:

```
//Añadir un evento al scope
scope.event = {
```

```
  date: (new Date()).getTime(),
  title: 'Entrega ejercicios sesión 1',
  contact: {
    name: 'Juan',
    middleName: 'Perez',
    lastName: 'Perez'
  }
};

//Crear nuestra plantilla
element = angular.element('<schedule-event event="event" is-today="true"
 delete-action="deleteEvent(event)" edit-action="editEvent(event)"></
schedule-event>');

//Compilar la plantilla
element = compile(element)(scope);
scope.$apply();
```

Una batería de tests para esta directiva podría ser la siguiente, donde iremos probando distintas combinaciones de atributos para ver si hace lo que queremos (`test/directive/scheduleEventSpec.js`):

```
'use strict';

describe('directive: scheduleEvent', function () {
  var element;
  var scope;
  var compile;

  beforeEach(function(){
    module('directives.schedule')
  });

  beforeEach(inject(function ($compile, $rootScope) {
    scope = $rootScope;
    compile = $compile;

    scope.deleteEvent = function () {
      console.log('deleting event');
    };
  }));

  it('should show a today event', function () {
    scope.event = {
      date: (new Date()).getTime(),
      title: 'Entregar los ejercicios de la sesión 1',
      contact: {
        name: 'Juan',
        middleName: 'Perez',
        lastName: 'Perez'
      }
    };

    element = angular.element('<schedule-event event="event" is-
today="true"></schedule-event>');
    element = compile(element)(scope);
```

```
    scope.$apply();
    var i = element.find('i');
    expect(i.hasClass('ion-ios7-time-outline')).toBe(true);
  });

  it('should show a future event', function () {
    scope.event = {
      date: (new Date()).getTime() + 86400000, //+2 days
      title: 'JPA 3,4',
      contact: {
        name: 'Juan',
        middleName: 'Perez',
        lastName: 'Perez'
      }
    };

    element = angular.element('<schedule-event event="event" is-
today="false"></schedule-event>');
    element = compile(element)(scope);
    scope.$apply();
    var i = element.find('i');
    expect(i.hasClass('ion-ios7-calendar-outline')).toBe(true);
  });

  it('should show the contact block', function () {
    scope.event = {
      date: (new Date()).getTime(),
      title: 'Entrega ejercicios sesión 1',
      contact: {
        name: 'Juan',
        middleName: 'Perez',
        lastName: 'Perez'
      }
    };

    element = angular.element('<schedule-event event="event" hide-
contact="false" is-today="true" delete-action="deleteEvent(event)" edit-
action="editEvent(event)"></schedule-event>');
    element = compile(element)(scope);
    scope.$apply();

    expect(element.find('p')[0]).not.toBeUndefined();
    expect(element.text()).toContain('Juan Perez Perez');
    expect(element.text()).toContain('Entrega ejercicios sesión 1');
  });

  it('should\'t show the contact block', function () {
    scope.event = {
      date: (new Date()).getTime(),
      title: 'Entrega ejercicios sesión 1',
      contact: null
    };

    element = angular.element('<schedule-event event="event"
 is-today="true" delete-action="deleteEvent(event)" edit-
action="editEvent(event)"></schedule-event>');
    element = compile(element)(scope);
    scope.$apply();
```

```
    expect(element.find('p')[0]).toBeUndefined();
    expect(element.text()).toContain('Entrega ejercicios sesión 1');
  });

  it('should\'t show the contact block despite it has a contact', function
 () {
    scope.event = {
      date: (new Date()).getTime(),
      title: 'Entrega ejercicios sesión 1',
      contact: {
        name: 'Juan',
        middleName: 'Perez',
        lastName: 'Perez'
      }
    };

    element = angular.element('<schedule-event event="event"
 is-today="true" delete-action="deleteEvent(event)" edit-
action="editEvent(event)" hide-contact="true"></schedule-event>');
    element = compile(element)(scope);
    scope.$apply();

    expect(element.find('p')[0]).toBeUndefined();
    expect(element.text()).toContain('Entrega ejercicios sesión 1');
  });

  it('should trigger a delete event', function () {
    spyOn(scope, 'deleteEvent');

    scope.event = {
      date: (new Date()).getTime(),
      title: 'Entrega ejercicios sesión 1',
      contact: {
        name: 'Juan',
        middleName: 'Perez',
        lastName: 'Perez'
      }
    };

    element = angular.element('<schedule-event event="event"
 is-today="true" delete-action="deleteEvent(event)" edit-
action="editEvent(event)"></schedule-event>');
    element = compile(element)(scope);
    scope.$apply();

    var editBtn = angular.element(element.find('a')[0]);
    editBtn.triggerHandler('click');
    scope.$apply();
    expect(scope.deleteEvent).toHaveBeenCalled();
  });
});
```

Cabe destacar el último test, donde utilizamos un *spy*, una funcionalidad de Jasmine que nos permite determinar si una función en concreto ha sido llamada.

12.4. Automatizando tareas con Grunt. Diseñando nuestro *workflow*

Ahora vamos a ver lo útil que puede resultarnos grunt para automatizar una serie de tareas. Para ello, crearemos un fichero llamado `Gruntfile.js` en la raíz de nuestro proyecto, que inicialmente será el siguiente:

```
module.exports = function (grunt) {
  // load all grunt tasks matching the `grunt-*` pattern
  require('load-grunt-tasks')(grunt);

  grunt.initConfig({});
  grunt.registerTask('default', []);
}
```

En él, ya hemos introducido un módulo, llamado `load-grunt-tasks`, que nos permite cargar de manera más cómoda el resto de módulos que incluyamos en nuestro fichero.

Verificación de código

Dado que JavaScript es un lenguaje tan permisivo, siempre es importante establecer unas convenciones de código. Es ahí donde entra JSHint[90], una herramienta *open source* que detecta errores y problemas potenciales en nuestro código JavaScript, y establece una serie de convenciones. Es muy restrictivo, y podemos relajarlo en base a nuestras necesidades y las de nuestro proyecto.

Para configurar JSHint en nuestro proyecto, modificaremos el fichero `Gruntfile.js` de la siguiente manera:

```
module.exports = function (grunt) {
  // load all grunt tasks matching the `grunt-*` pattern
  require('load-grunt-tasks')(grunt);

  grunt.initConfig({
    'jshint' : {
      options: {
        curly: true,
        eqeqeq: true,
        eqnull: true,
        browser: true,
        globals: {
          jQuery: true
        },
      },
      default : ['src/**/*.js']
    }
  });

  grunt.registerTask('default', ['jshint']);
}
```

[90] http://jshint.com/

Si ahora lanzamos el comando `grunt` en nuestra terminal, se ejecutará la tarea *default*, que realiza la validación de todos los ficheros con extensión `.js` en alguna de las subcarpetas de `src`.

Veremos que nos da error en el fichero `src/controller/AccessController.js` porque hay un par de líneas que no hemos finalizado con punto y coma. También, nos dirá que hay una sentencia `if` en el fichero `src/filters/textOrDefault.js` que no tiene llaves

Una vez corregidos estos dos errores, la tarea se ejecutará correctamente.

Testing

Una vez verificado el código, haremos que los tests se lancen automáticamente con karma. Para ello, añadiremos karma a nuestro *workflow*:

```
module.exports = function (grunt) {
  //...

  grunt.initConfig({
    'jshint' : {
      //...
    },

    'karma' : {
      'default' : {
        'configFile' : 'karma.conf.js',
        'options': {
          singleRun: true
        }
      }
    }
  });

  grunt.registerTask('default', ['jshint', 'karma']);
}
```

Como véis, hemos sobreescrito la opción `singleRun` para asegurarnos de que no se queda a la espera de cambios para volver a lanzar la batería de tests.

Ahora, al lanzar `grunt` se ejecutará JSHint y, si pasa correctamente, se lanzarán después los tests que habíamos hecho con karma.

Generando código de distribución

En tiempo de desarrollo, es muy cómodo y recomendable tener varios ficheros de código fuente. Sin embargo, a la hora de ir a producción, lo normal es tener un único fichero fuente con todo el código, ya sea minificado o no. El mismo angularJS, como podemos ver en su GitHub[91], tiene un sinfín de ficheros pero nosotros únicamente importamos el fichero `angular.js` o `angular.min.js`. Esto se realiza de una manera sencilla con los plugins `grunt-contrib-concat` y `grunt-contrib-uglify`. El primero de ellos se encargará de concatenar todos los ficheros en uno solo, mientras que el segundo utilizará este resultado para generar un fichero minificado.

[91] https://github.com/angular/angular/tree/master/modules

Como esto no lo realizaremos siempre, registraremos una tarea, que llamaremos `dist`, que realizará esta labor. Es muy importante que los tests pasen correctamente antes de generar un fichero de distribución, con lo que repetiremos las vistas anteriormente.

```
module.exports = function (grunt) {
  // ...

  grunt.initConfig({
    'pkg': grunt.file.readJSON('package.json'),

    'jshint' : {
      // ...
    },

    'karma' : {
      // ...
    },

    'concat': {
      'dist' : {
        'src' : ['src/**/*.js'],
        'dest': 'dist/<%=pkg.name%>-<%=pkg.version%>.js'
      }
    },

    'uglify': {
      'options': {
        'mangle':false
      },
      'dist':{
        'files': {
          'dist/<%=pkg.name%>-<%=pkg.version%>.min.js' : ['dist/<
%=pkg.name%>-<%=pkg.version%>.js']
        }
      }
    }
  });

  grunt.registerTask('default', ['jshint', 'karma']);
  grunt.registerTask('dist',
 ['jshint', 'karma', 'concat:dist', 'uglify:dist']);
}
```

Lanzando el comando `grunt dist`, veremos que se ejecuta todo, y finalmente se habrá creado una carpeta `dist` con dos nuevos ficheros:

```
.
├── Gruntfile.js
├── bower_components
├── dist
│   ├── angular-automation-testing-0.0.0.js
│   └── angular-automation-testing-0.0.0.min.js
├── karma.conf.js
├── node_modules
├── src
```

```
│   ├── controller
│   ├── directive
│   ├── filters
│   └── providers
└── test
    ├── controller
    ├── directive
    ├── filters
    └── providers
```

Observando cambios para lanzar tests

Otra tarea muy interesante que podemos programar es que los tests se lancen automáticamente tan pronto salvemos los cambios de algún fichero javascript. Para ello nos valdremos de la ayuda del plugin `grunt-contrib-watch`.

```
module.exports = function (grunt) {
  // ...

  grunt.initConfig({

    // ...

    watch: {
      scripts: {
        files: ['src/**/*.js', 'test/**/*.js'],
        tasks: ['jshint', 'karma'],
        options: {
          spawn: false,
        },
      },
    },
  });

  grunt.registerTask('default', ['jshint', 'karma']);
  grunt.registerTask('dist',
 ['jshint', 'karma', 'concat:dist', 'uglify:dist']);
}
```

Lanzando ahora el comando `grunt watch`, veremos que la terminal se pone en espera. Modificando cualquier fichero, vemos cómo se registra el cambio y se lanzan los tests.

Otros plugins de utilidad

Hemos visto unos cuantos plugins que son de gran utilidad para nuestros, y ampliamente usados.

Otros plugins interesantes podrían ser:

- `grunt-angular-injector`[92]: modifica nuestro código AngularJS para ayudarnos a solventar el problema de la minificación de código y la inyección de dependencias.

[92] https://github.com/alexgorbatchev/grunt-angular-injector

- `grunt-html2js`[93]: introduce en el servicio `$templateCache` todas nuestras vistas. De esta manera, ya están cacheadas y no se tienen que pedir por AJAX.
- `grunt-groundskeeper`[94]: elimina los `console.log` y `debugger` de nuestro código.
- `grunt-contrib-cssmin`[95]: al igual que minificamos javascript, también podemos hacerlo con nuestros CSS.
- `grunt-contrib-less`[96]: compila nuestro código LESS a CSS.
- `grunt-contrib-imagemin`[97]: optimiza nuestras imágenes para un entorno web.
- `grunt-contrib-htmlmin`[98]: minifica nuestro código HTML.
- `grunt-open`[99]: podría usarse, por ejemplo, para abrir el navegador una vez generado el código.
- `grunt-concurrent`[100]: nos permite lanzar tareas de grunt de manera concurrente.
- `grunt-conventional-changelog`[101]: genera un *changelog* a partir de los metadatos de Git.

12.5. Un paso más allá

Hoy en día tenemos una gran cantidad de servidores de integración continua que nos permiten realizar estas tareas automáticas una vez hemos hecho push en nuestro repositorio. Travis[102], por ejemplo, nos da este servicio de manera gratuita para proyectos *open source*. Podemos generar un *hook* que lanza los tests y genera el código de distribución, y luego despliega *releases* en el repositorio de nuestro proyecto.

93 https://www.npmjs.com/package/grunt-html2js
94 https://github.com/Couto/grunt-groundskeeper
95 https://www.npmjs.com/package/grunt-contrib-cssmin
96 https://www.npmjs.com/package/grunt-contrib-less
97 https://www.npmjs.com/package/grunt-contrib-imagemin
98 https://www.npmjs.com/package/grunt-contrib-htmlmin
99 https://github.com/jsoverson/grunt-open
100 https://github.com/sindresorhus/grunt-concurrent
101 https://github.com/btford/grunt-conventional-changelog
102 https://travis-ci.org/